AFIT/GCS/ENG/93D-12

D

PERFORMANCE MEASUREMENT OF THREE COMMERCIAL OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS

THESIS

Timothy J. Halloran
Captain, USAF

AFIT/GCS/ENG/93D-12

Approved for public [illegible] ...ited

AFIT/GCS/ENG/93D-12

PERFORMANCE MEASUREMENT OF THREE COMMERCIAL OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Timothy J. Halloran, B.S.

Captain, USAF

December, 1993

Acknowledgements

There are several people I need to thank for their support of my efforts. First, I would like to thank Major Mark A. Roth, my thesis advisor, for his guidance and support. I also extend thanks to my committee members: Lieutenant Colonel Hobart, Dr Potoczny, and Dr Suter for their expertise and their time spent reviewing my thesis draft. I would also like to thank Itasca, Matisse, and ObjectStore customer support, and especially Mr Laurent Censier of Intellitic International and Mr Randal Zoeller of Itasca Systems Incorporated. All the vendors were helpful during this research. Finally, I would like to thank Dan Zambon and David Doak, my system administrators, who put up with all the software installations, software version updates, hardware configuration changes, and system crashes which seemed to shadow this research.

Timothy J. Halloran

Table of Contents

Page

Acknowledgements ii

List of Figures ix

List of Tables xiii

Abstract xviii

I. Introduction 1

1.1 Background 1

1.2 Problem Statement 3

1.3 Objectives 3

1.4 Methodology 3

1.5 Materials and Equipment 5

1.6 Sequence of Presentation 6

II. Literature Review 7

2.1 DBMS Benchmarks 7

2.2 DBMS Benchmark Criteria 7

2.3 DBMS Benchmark Standards Organizations 9

2.4 Benchmarks for OLTP and Relational DBMSs 9

2.4.1 TPC-A and TPC-B 10

2.4.2 TPC-C 14

2.4.3 The Wisconsin Benchmark 17

2.5 Benchmarks for Object-Oriented DBMSs 18

2.5.1 Simple Database Operations Benchmark 18

2.5.2 HyperModel Benchmark 20

Page

2.5.3 Object Operations Version 1 (OO1) 22

2.5.4 OO7 24

2.6 Computer Simulation 26

2.7 Current Simulation Systems Using Object-Oriented DBMSs . . . 26

2.7.1 Visual Intelligence and Electronic Warfare Simulation Workbench 26

2.7.2 Saber Wargame 27

2.8 Simulation Environments 28

2.8.1 Post-Processing and Simulation Graphics 29

2.8.2 The Joint Modeling and Simulation System 30

2.8.3 J-MASS Data Management 31

2.9 Summary 32

III. Object-Oriented DBMS Functional Comparison 33

3.1 DBMS Overview 33

3.2 Transaction Properties 34

3.2.1 ACID Requirements 34

3.2.2 Long Transactions 34

3.2.3 Nested Transactions 36

3.2.4 Non-Blocking Read-Only Transactions 36

3.3 Locking and Concurrency Control 37

3.4 Security Authorization 37

3.5 Query Capability 37

3.5.1 Itasca 38

3.5.2 ObjectStore 38

3.6 Distributed Database Capabilities 39

3.7 Programming-Language Integration 39

3.7.1 Itasca 39

Page

3.7.2 Matisse 40

3.7.3 ObjectStore 40

3.8 Database Architecture 40

3.8.1 Itasca 41

3.8.2 Matisse 41

3.8.3 ObjectStore 42

3.9 Summary 43

IV. OO1 Benchmark Analysis, Design, and Implementation 44

4.1 Object-Oriented Analysis 44

4.1.1 Identify Classes and Objects 44

4.1.2 Identify Structures 49

4.1.3 Identify Subjects 49

4.1.4 Define Attributes 49

4.1.5 Define Services 50

4.2 Object-Oriented Design 52

4.2.1 Design of the Human Interaction Component 53

4.2.2 Design of the Problem Domain Component 53

4.3 Benchmark Implementations 56

4.3.1 Itasca Implementation 56

4.3.2 Matisse Implementation 61

4.3.3 ObjectStore Implementation 64

4.4 Verification of Benchmark Program Correctness 66

4.5 Summary 66

V. Simulation Benchmark 68

5.1 Benchmark Description 68

5.1.1 Benchmark Requirements 68

Page

5.1.2 Benchmark Measures 74

5.1.3 Benchmark Justification 75

5.2 Simulation Benchmark Design 77

5.2.1 Design of the Human Interaction Component 77

5.2.2 Design of the Problem Domain Component 81

5.2.3 Design of the Database Interface 84

5.3 Simulation Benchmark Implementations 84

5.3.1 ObjectStore Implementation 85

5.3.2 Non-Persistent Implementation 85

5.3.3 Itasca and Matisse 85

5.3.4 Multiple Object Model Problem 85

5.4 Summary 87

VI. Benchmark Results Analysis 88

6.1 OO1 Benchmark Results 88

6.1.1 Small Remote Database Results 89

6.1.2 Small Local Database Results 91

6.1.3 Small Remote Database (NLOR) Results 93

6.1.4 Small Local Database (NLOR) Results 95

6.1.5 Large Remote Database Results 95

6.1.6 Large Local Database Results 99

6.1.7 Benchmark Database Load Times 102

6.1.8 Matisse Version Collection Results 102

6.1.9 Reverse Traversal Results 102

6.2 Simulation Benchmark Results 104

6.2.1 Quantitative Results 105

6.2.2 Qualitative Results 105

6.3 Summary 107

Page

VII. Conclusions and Recommendations 108

7.1 Overview 108

7.2 Conclusions 108

7.3 Lessons Learned 109

7.4 Recommendations 110

7.5 Summary 111

Appendix A. Object-Oriented Analysis and Design Notation Summary 112

Appendix B. Detailed OO1 Benchmark Results 114

B.1 OO1 Benchmark Results for the Itasca DBMS 117

B.2 OO1 Benchmark Results for the Matisse DBMS 134

B.3 OO1 Benchmark Results for the ObjectStore DBMS 168

Appendix C. Statistical Analysis of the OO1 Benchmark Results 197

Appendix D. Detailed Simulation Benchmark Results 208

D.0.1 Simulation Benchmark Results for the ObjectStore DBMS 209

D.0.2 Non-Persistent Simulation Benchmark Results 214

Appendix E. Statistical Analysis of the Simulation Benchmark Results 217

Appendix F. Benchmark Support Library 219

F.1 The Stopwatch 219

F.1.1 stopwtch.hh 220

F.1.2 stopwtch.cc 221

F.2 The Dice Module 222

F.2.1 dice.hh 222

F.2.2 dice.cc 223

F.3 The Expon Module 224

Page

F.3.1 expon.hh 224

F.3.2 expon.cc 224

F.4 A U(0,1) Random Number Generator 225

F.4.1 pmmlcg.h 225

F.4.2 pmmlcg.c 226

Appendix G. Code Style Guide 229

G.1 Code Indentation and Spacing 229

G.2 Naming Conventions 229

G.2.1 Variable and Function Names 229

G.2.2 Constant Names 229

G.3 Comments 230

G.3.1 Module/Header Comments 230

G.3.2 Inline Comments 230

G.3.3 Function Headers and Separator Comments 230

G.4 Error Output 232

G.5 Debug Output 232

G.5.1 debug.hh 233

Appendix H. Benchmark Source Code 236

Bibliography 237

Vita 240

List of Figures

Figure		Page
1.	TPC-A and TPC-B Entity/Relationship Diagram	11
2.	TPC-A and TPC-B Transaction Profile	12
3.	Evolution of Object-Oriented DBMS Benchmarks	19
4.	Remote or Client-Server Database Configuration	24
5.	Three-Level Hierarchy of a Simulation System	29
6.	Components of a Data-Driven Simulator	30
7.	A Sample Itasca Query	38
8.	A Sample ObjectStore DML Query	39
9.	OOA Diagram for the OO1 Benchmark	45
10.	Parts and Connections in the OO1 Benchmark Database	47
11.	Part and Connection Representation in a Coad/Yourdon OOA Diagram	48
12.	Part and Connection Representation in a Rumbaugh Object Model	48
13.	OOD Diagram for the OO1 Benchmark—Subject Layer	54
14.	OOD Diagram for the OO1 Benchmark	55
15.	Itasca OOD Diagram for the OO1 Benchmark	57
16.	Event Graph (Queuing Model) for the Simulation Benchmark	72
17.	Simulation Benchmark Design—Big Picture	77
18.	Simulation Benchmark User Interface Menu	78
19.	Example of the Summary Report	80
20.	Simulation Benchmark Problem Domain OOD Diagram	82
21.	The Simulation Benchmark Program (ObjectStore Version)	86
22.	OO1 Benchmark Results for *Small Remote* Database	90
23.	OO1 Benchmark Results for *Small Local* Database	92
24.	OO1 Benchmark Results for *Small Remote* Database (NLOR)	94
25.	OO1 Benchmark Results for *Small Local* Database (NLOR)	96

Figure Page

26. OO1 Benchmark Results for *Large Remote* Database 100

27. OO1 Benchmark Results for *Large Local* Database 101

28. Simulation Benchmark Hour Run Results by Time Slice Setting for *Small* Database 106

29. Coad/Yourdon OOA/OOD Notations 113

30. Itasca OO1 Average Benchmark Results for *Small Remote* Database . . . 119

31. Itasca OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Remote* Database 121

32. Itasca OO1 Average Benchmark Results for *Small Local* Database 123

33. Itasca OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database 125

34. Itasca OO1 Average Benchmark Results for *Small Remote* Database (NLOR) 127

35. Itasca OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Remote* Database (NLOR) 129

36. Itasca OO1 Average Benchmark Results for *Small Local* Database (NLOR) 131

37. Itasca OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database (NLOR) 133

38. Matisse OO1 Average Benchmark Results for *Small Remote* Database . . 137

39. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Remote* Database 139

40. Matisse OO1 Average Benchmark Results for *Small Local* Database 141

41. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database 143

42. Matisse OO1 Average Benchmark Results for *Small Local* Database 200 Second Delay for Version Collection 145

43. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database 200 Second Delay for Version Collection 147

44. Matisse OO1 Average Benchmark Results for *Small Remote* Database (NLOR) 149

Figure Page

45. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Remote* Database (NLOR) 151

46. Matisse OO1 Average Benchmark Results for *Small Remote* Database (NLOR) 200 Second Delay for Version Collection 153

47. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Remote* Database (NLOR) 200 Second Delay for Version Collection 155

48. Matisse OO1 Average Benchmark Results for *Small Local* Database (NLOR) 157

49. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database (NLOR) 159

50. Matisse OO1 Average Benchmark Results for *Large Remote* Database . . 161

51. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Large Remote* Database 163

52. Matisse OO1 Average Benchmark Results for *Large Local* Database 165

53. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Large Local* Database 167

54. ObjectStore OO1 Average Benchmark Results for *Small Remote* Database 170

55. ObjectStore OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Remote* Database 172

56. ObjectStore OO1 Average Benchmark Results for *Small Local* Database . 174

57. ObjectStore OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database 176

58. ObjectStore OO1 Average Benchmark Results for *Small Local* Database (second) 178

59. ObjectStore OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database (second) 180

60. ObjectStore OO1 Average Benchmark Results for *Small Remote* Database (NLOR) 182

61. ObjectStore OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Remote* Database (NLOR) 184

Figure Page

62. ObjectStore OO1 Average Benchmark Results for *Small Local* Database (NLOR) 186

63. ObjectStore OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database (NLOR) 188

64. ObjectStore OO1 Average Benchmark Results for *Large Remote* Database 190

65. ObjectStore OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Large Remote* Database 192

66. ObjectStore OO1 Average Benchmark Results for *Large Local* Database . 194

67. ObjectStore OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Large Local* Database 196

68. ObjectStore Simulation Benchmark Hour Run Results by Time Slice Setting for *Small Remote* Database 211

69. ObjectStore Simulation Benchmark Hour Run Results by Time Slice Setting for *Small Local* Database 213

70. Non-Persistent Simulation Benchmark Hour Run Results by Time Slice Setting for *Small* Database 216

71. OOA Diagram for the Benchmark Support Library 220

72. Example of Source Code Indentation and Spacing 230

73. Source File Comment Block Example 231

74. Examples of Program Error Output 233

75. Example of Debug Output 233

List of Tables

Table Page

1. DBMS Benchmark Summary 26
2. Estimated 1993 Market Share of Object-Oriented DBMS Companies . . . 34
3. Summary of Object-Oriented DBMS Capabilities 42
4. AFIT Benchmark System Configuration 89
5. OO1 Benchmark Results for *Small Remote* Database 90
6. OO1 Benchmark Results for *Small Local* Database 92
7. OO1 Benchmark Results for *Small Remote* Database (NLOR) 93
8. OO1 Benchmark Results for *Small Local* Database (NLOR) 95
9. Summary of OO1 Benchmark *Small* Database Results 97
10. OO1 Benchmark Results for *Large Remote* Database 99
11. OO1 Benchmark Results for *Large Local* Database 99
12. Summary of OO1 Benchmark *Large* Database Results 103
13. OO1 Benchmark Load Times for *Small* Database 103
14. OO1 Benchmark Load Times for *Large* Database 103
15. Summary of OO1 Benchmark Results 104
16. Simulation Benchmark Results for *Small* Database 106
17. OO1 Benchmark Load Times for *Small* Database 115
18. OO1 Benchmark Load Times for *Large* Database 115
19. Itasca OO1 Summary Results for *Small Remote* Database 118
20. Itasca OO1 Normalized Reverse Traversal Results for *Small Remote* Database 119
21. Itasca OO1 Raw Benchmark Results for *Small Remote* Database 120
22. Itasca OO1 Summary Results for *Small Local* Database 122
23. Itasca OO1 Normalized Reverse Traversal Results for *Small Local* Database 123
24. Itasca OO1 Raw Benchmark Results for *Small Local* Database 124
25. Itasca OO1 Summary Results for *Small Remote* Database (NLOR) 126

Table Page

26. Itasca OO1 Normalized Reverse Traversal Results for *Small Remote* Database (NLOR) . . . 127

27. Itasca OO1 Raw Benchmark Results for *Small Remote* Database (NLOR) 128

28. Itasca OO1 Summary Results for *Small Local* Database (NLOR) 130

29. Itasca OO1 Raw Benchmark Results for *Small Local* Database (NLOR) . 132

30. Matisse OO1 Summary Results for *Small Remote* Database 136

31. Matisse OO1 Normalized Reverse Traversal Results for *Small Remote* Database . . . 137

32. Matisse OO1 Raw Benchmark Results for *Small Remote* Database 138

33. Matisse OO1 Summary Results for *Small Local* Database 140

34. Matisse OO1 Normalized Reverse Traversal Results for *Small Local* Database 141

35. Matisse OO1 Raw Benchmark Results for *Small Local* Database 142

36. Matisse OO1 Summary Results for *Small Local* Database 200 Second Delay for Version Collection . . . 144

37. Matisse OO1 Normalized Reverse Traversal Results for *Small Local* Database 200 Second Delay for Version Collection . . . 145

38. Matisse OO1 Raw Benchmark Results for *Small Local* Database 200 Second Delay for Version Collection . . . 146

39. Matisse OO1 Summary Results for *Small Remote* Database (NLOR) . . . 148

40. Matisse OO1 Normalized Reverse Traversal Results for *Small Remote* Database (NLOR) . . . 149

41. Matisse OO1 Raw Benchmark Results for *Small Remote* Database (NLOR) 150

42. Matisse OO1 Summary Results for *Small Remote* Database (NLOR) 200 Second Delay for Version Collection . . . 152

43. Matisse OO1 Normalized Reverse Traversal Results for *Small Remote* Database (NLOR) 200 Second Delay for Version Collection . . . 153

44. Matisse OO1 Raw Benchmark Results for *Small Remote* Database (NLOR) 200 Second Delay for Version Collection . . . 154

45. Matisse OO1 Summary Results for *Small Local* Database (NLOR) 156

Table | Page

46. Matisse OO1 Normalized Reverse Traversal Results for *Small Local* Database (NLOR) 157

47. Matisse OO1 Raw Benchmark Results for *Small Local* Database (NLOR) . 158

48. Matisse OO1 Summary Results for *Large Remote* Database 160

49. Matisse OO1 Normalized Reverse Traversal Results for *Large Remote* Database 161

50. Matisse OO1 Raw Benchmark Results for *Large Remote* Database 162

51. Matisse OO1 Summary Results for *Large Local* Database 164

52. Matisse OO1 Normalized Reverse Traversal Results for *Large Local* Database 165

53. Matisse OO1 Raw Benchmark Results for *Large Local* Database 166

54. ObjectStore OO1 Summary Results for *Small Remote* Database 169

55. ObjectStore OO1 Normalized Reverse Traversal Results for *Small Remote* Database 170

56. ObjectStore OO1 Raw Benchmark Results for *Small Remote* Database . . 171

57. ObjectStore OO1 Summary Results for *Small Local* Database 173

58. ObjectStore OO1 Normalized Reverse Traversal Results for *Small Local* Database 174

59. ObjectStore OO1 Raw Benchmark Results for *Small Local* Database . . . 175

60. ObjectStore OO1 Summary Results for *Small Local* Database (second) . . 177

61. ObjectStore OO1 Normalized Reverse Traversal Results for *Small Local* Database (second) 178

62. ObjectStore OO1 Raw Benchmark Results for *Small Local* Database (second) 179

63. ObjectStore OO1 Summary Results for *Small Remote* Database (NLOR) . 181

64. ObjectStore OO1 Normalized Reverse Traversal Results for *Small Remote* Database (NLOR) 182

65. ObjectStore OO1 Raw Benchmark Results for *Small Remote* Database (NLOR) 183

66. ObjectStore OO1 Summary Results for *Small Local* Database (NLOR) . . 185

67. ObjectStore OO1 Normalized Reverse Traversal Results for *Small Local* Database (NLOR) 186

Table		Page
68.	ObjectStore OO1 Raw Benchmark Results for *Small Local* Database (NLOR)	187
69.	ObjectStore OO1 Summary Results for *Large Remote* Database	189
70.	ObjectStore OO1 Normalized Reverse Traversal Results for *Large Remote* Database	190
71.	ObjectStore OO1 Raw Benchmark Results for *Large Remote* Database	191
72.	ObjectStore OO1 Summary Results for *Large Local* Database	193
73.	ObjectStore OO1 Normalized Reverse Traversal Results for *Large Local* Database	194
74.	ObjectStore OO1 Raw Benchmark Results for *Large Local* Database	195
75.	Small-Sample Test Statistic for OO1 Benchmark *Small Remote* Database Results	199
76.	Small-Sample Test Statistic for OO1 Benchmark *Small Local* Database Results	200
77.	Small-Sample Test Statistic for OO1 Benchmark *Small Remote* Database Results (NLOR)	201
78.	Small-Sample Test Statistic for OO1 Benchmark *Small Local* Database Results (NLOR)	202
79.	Small-Sample Test Statistic for OO1 Benchmark *Large Remote* Database Results	203
80.	Small-Sample Test Statistic for OO1 Benchmark *Large Local* Database Results	204
81.	Small-Sample Test Statistic for OO1 Benchmark Matisse Results (No Delay/300 Second Delay)	205
82.	Small-Sample Test Statistic for OO1 Benchmark Matisse *Local/Remote* Database Results	206
83.	Small-Sample Test Statistic for OO1 Benchmark ObjectStore *Local/Remote* Database Results	207
84.	ObjectStore Simulation Benchmark Summary Results for *Small Remote* Database	210
85.	ObjectStore Simulation Benchmark Throughput Results for *Small Remote* Database	211

Table Page

86. ObjectStore Simulation Benchmark Summary Results for *Small Local* Database . . . 212

87. ObjectStore Simulation Benchmark Throughput Results for *Small Local* Database . . . 213

88. Non-Persistent Simulation Benchmark Summary Results for *Small* Database 215

89. Non-Persistent Simulation Benchmark Throughput Results for *Small* Database . . . 216

90. Small-Sample Test Statistic for Simulation Benchmark *Small* Database Results . . . 218

91. Standard Abbreviations . . . 232

AFIT/GCS/ENG/93D-12

Abstract

The goal of this thesis was to study the performance of three commercial object-oriented database management systems. The commercial systems studied included: Itasca, sold by Itasca Systems Incorporated; Matisse, sold by Intellitic International; and ObjectStore, sold by Object Design Incorporated. To examine performance of these database management systems two benchmarks were run: the OO1 benchmark and a new AFIT Simulation benchmark. The OO1 benchmark was designed, implemented, and run on all three database management systems. ObjectStore was our top performer on all configurations of the OO1 benchmark. The AFIT Simulation benchmark was designed, implemented, and run on the ObjectStore database management system. A non-persistent version of the benchmark was also created in the C++ programming language. There was minimal performance overhead incurred due to the use of ObjectStore, especially when compared to the functional benefits gained. We concluded that there are major differences between the performance levels offered in current commercial object-oriented database management systems. We also concluded that a programming language interface to an object-oriented database management system should not be middle ground. Either it should be closely tied to a specific language or not tied to a specific language at all.

PERFORMANCE MEASUREMENT OF THREE COMMERCIAL OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS

I. Introduction

1.1 Background

The next generation of database management systems, object-oriented database management systems, is starting to arrive on the market today. The various implementations of these systems are incredibly diverse, much more diverse than relational database systems that have been dominant since the mid-1980s. All relational database management systems are based on the relational data model. The relational data model is uniformly used in different database implementations and is firmly based in mathematics (set theory and first order predicate logic) [12]. In contrast, the emerging object-oriented data model is neither uniformly implemented nor firmly based in mathematics. In fact, exactly what traits are required for a database to claim that it is object-oriented is still under some debate [2, 37]. This lack of agreement has created diversity in the capabilities of today's object-oriented database management systems.

Despite this diversity, object-oriented DBMSs are indeed useful today. They are bringing DBMS functionality to applications which traditionally have used only custom file-based storage systems. Engineering applications, such as Computer Aided Design (CAD) and Computer Aided Publishing (CAP), and also computer simulation, have not widely used existing commercial DBMSs for several reasons. The most critical reason is *performance*. Interactive engineering applications require database systems which are ten to one hundred times faster than traditional DBMSs [9]. Some object-oriented DBMSs can provide this much-needed level of performance.

Because performance is a critical requirement, it is imperative to be able to measure the performance of object-oriented DBMSs. It is also necessary to focus performance measurement on those object-oriented DBMS services which are the most critical. To identify which services are the most critical, the applications which use object-oriented DBMSs must be investigated. Once critical services have been identified, a benchmark can be used as a tool to measure object-oriented DBMS performance.

A *benchmark* is a program used to quantitatively measure the performance of any software or hardware system. Gray notes that a benchmark can be thought of as a *workload* [14:1]. The hardware or software on which the benchmark is run is called the *system under test*. The performance measure of a system under test on a benchmark could be a time to completion (seconds) or a throughput metric (work/seconds). The performance measurement can be combined with the price of the system under test to give a price/performance ratio.

Good performance of a system under test on a benchmark does not indicate that the system will perform well on every type of application. The benchmark is only a valid yardstick for applications which are similar to the benchmark. For example, if you are planning to build a software system which performs a great deal of floating-point operations, then examining the results of floating-point operations benchmarks can aid in your selection of a computer system. But if you are planning to do word processing on the computer system, then the results of floating-point operations benchmarks will be useless.

Benchmarks allow comparison of different systems for an application without actually having to build the application on all the different systems under consideration. When an application is going to be very large, it is often impossible to build the complete application to test different systems. Therefore, an important property of benchmarks is that they be small, and thus reasonably simple to implement. To date, there are four existing object-oriented DBMSs benchmarks:

- Simple Database Operations
- HyperModel
- Object Operations Version 1 (OO1)
- OO7

The Simple Database Operations benchmark uses a database of authors and books to measure performance on "simple, object-oriented queries that engineering database applications perform" [35:387]. The HyperModel benchmark is a more complex version of the Simple Database Operations benchmark [15]. The OO1 benchmark is a simpler, more focused, version of the Simple Database Operations benchmark [9]. The OO7 benchmark is a new benchmark developed at the University of Wisconsin which attempts to be a more complete measure of database performance than the OO1 benchmark [7].

1.2 Problem Statement

The problem was that it was not known if the performance of the object-oriented DBMSs available at AFIT was good enough to be used for research applications, especially research in computer simulation. To determine this, it was necessary to benchmark the performance of the three commercial object-oriented DBMSs available at AFIT. This problem was complicated by the diversity of object-oriented DBMS interfaces, the complexity of the object-oriented DBMSs, the wide variety of applications to which object-oriented DBMSs can be applied, and the question of what specific services simulation applications require.

1.3 Objectives

The primary objective of this thesis was to measure the performance and functionality of the three commercial object-oriented DBMS available at AFIT.

A second objective was to create a new or extended benchmark for simulation applications. This was necessary to provide simulation research projects a yardstick to evaluate if any of the object-oriented DBMS available at AFIT would be useful to them.

1.4 Methodology

This research was conducted in four stages. In each stage the following three commercial object-oriented DBMSs were tested:

- Itasca

- Matisse
- ObjectStore

These three object-oriented DBMSs were selected because we believe that they represent a reasonable cross-section of commercial object-oriented DBMS industry.

Itasca, Matisse, and ObjectStore were used in the following four stages of research:

1. Functional comparison of the three commercial object-oriented DBMSs.
2. Running the OO1 benchmark on the three commercial object-oriented DBMSs.
3. Creating a specification for an object-oriented DBMS benchmark for the simulation domain.
4. Running the simulation benchmark on the three commercial object-oriented DBMSs.

During stage one we investigated the functional capabilities of the Itasca, Matisse, and ObjectStore DBMSs. The goal was to determine the functional differences between the databases. Knowing the functional differences between the databases aided our analysis of the benchmark results obtained later in this research. A secondary reason for investigating the functional capabilities of the three commercial object-oriented DBMSs was to be able to gain enough practical knowledge about the databases to implement the benchmarks which were constructed in the following phases.

During stage two we created an implementation of the OO1 benchmark for the Itasca, Matisse, and ObjectStore DBMSs. To investigate the performance of Itasca, Matisse, and ObjectStore, we first wanted to investigate their performance on a standard, well defined benchmark. We selected the OO1 benchmark for the following reasons:

- *Maturity*: The OO1 benchmark is the most mature and completely specified of all the object-oriented DBMS benchmarks. The benchmark evolved from an earlier benchmark.
- *Industry Acceptance*: The OO1 benchmark has wide acceptance from vendors in the object-oriented DBMS industry, or at least we felt that was true at the start of this research.

- *Reasonable Implementation Effort*: To accomplish the implementation of a standard benchmark, and a new benchmark in the simulation domain, the standard benchmark must not require an unreasonable amount of time to implement.

Stage three involved creating a specification for a benchmark which would execute a computer simulation environment inside an object-oriented DBMS. We examined literature about DBMS benchmarks and current simulations to define a simple simulation. The benchmark was to be qualitative as well as quantitative. The benchmark investigated the ability of the three commercial object-oriented DBMSs to support computer simulation.

Stage four implemented the simulation defined in the previous stage. A complete implementation of the simulation benchmark was created for the Itasca, Matisse, and ObjectStore DBMSs.

1.5 Materials and Equipment

For this research, two Sun SPARCstation 2 workstations were set up as test computers. The SPARCstation 2 is a general purpose engineering workstation. One workstation, *prowler*, acted as the database server, and the other workstation, *doc*, as the client. The database server was configured with two additional disk drives, one to hold the DBMS software, and the other to hold the test databases. The benchmark runs were run during the evening to avoid heavy network traffic during tests.

For this research we used version 2.2 of the Itasca object-oriented DBMS. Itasca was originally developed as the Orion database system by Microelectronics and Computer Technology Corporation. The Orion database was enhanced and is now sold by Itasca Systems Incorporated of Minneapolis, Minnesota. We used version 2.2.0 of the Matisse object-oriented DBMS. Matisse was developed by Intellitic International of France. We also used version 2.0.1 of the ObjectStore object-oriented DBMS. ObjectStore was developed by Object Design Incorporated of Burlington, Massachusetts.

1.6 Sequence of Presentation

In Chapter II we present a review of pertinent literature in the area of DBMS benchmarking and computer simulation. Chapter III investigates the functional capabilities and differences between the Itasca, Matisse, and ObjectStore object-oriented DBMSs. In Chapter IV we describe the analysis, design, and implementation of the OO1 benchmark. We also examine the problems encountered when working with the three object-oriented DBMSs. Chapter V describes the simulation benchmark developed for this research and describes our implementation of this benchmark. In Chapter VI we examine our results from running the OO1 benchmark and the simulation benchmark, and in Chapter VII present some final conclusions and recommendations.

II. Literature Review

In this chapter we review literature about DBMS benchmarking and computer simulation. The review of DBMS benchmarks examines the capabilities of existing benchmarks. The review of computer simulation examines the use of object-oriented DBMSs in simulation environments today and the capabilities needed by simulation environments.

2.1 DBMS Benchmarks

DBMS benchmarks are a way to measure the performance and/or functionality of a DBMS. They can also be used to find the lowest-cost DBMS and computer system for a required job. The next four sections survey DBMS benchmarks. First, the criteria for a good DBMS benchmark is covered. Second, the role of the only DBMS benchmark standards organization, the Transaction Processing Performance Council, is examined. Then eight important DBMS benchmarks are looked at. For each benchmark we point out the important strengths and weaknesses of the benchmark. Included are benchmarks for on line transaction processing (OLTP), relational, and object-oriented DBMSs. The following items about each benchmark are examined:

- The benchmark problem domain
- The benchmark database
- The benchmark operations
- The measurements (or results) of the benchmark

For more detailed information about any specific benchmark, the source documents on that benchmark should be examined.

2.2 DBMS Benchmark Criteria

DBMS benchmarks are *domain-specific benchmarks*. These benchmarks attempt to quantitatively measure the performance of a DBMS in a specific domain area, such as decision support or OLTP. In [14] Gray proposes the following criteria for a good domain-specific benchmark: *relevant*, *portable*, *scalable*, and *simple*.

A benchmark must be *relevant* to be a useful yardstick for DBMS performance. For example, if you are planning to use a DBMS for OLTP, then results of OLTP benchmarks can aid in selection of a DBMS and a computer system. But if you are planning to use the database for a decision support system, then OLTP benchmarks are not useful because they are not relevant to the decision support domain.

A benchmark must be *portable* so that it can be run on several different DBMSs and computer systems. Ideally, a benchmark should be able to be run on all the DBMSs which support the domain (e.g., OLTP) for which it measures performance.

A benchmark should be *scalable* to large and small computer systems. As the capabilities of the computer system increase, the benchmark should "scale-up" to credibly measure the performance of that computer system. Gray also notes that a benchmark should scale-up to new types of computer systems (e.g. parallel computer systems) as "computer performance and architecture evolve" [14:5].

A benchmark should be *simple* so it can be understood and easily implemented. If a benchmark is as complex as your intended application, then there would be little point to using the benchmark (your application could be used to measure DBMS performance). A benchmark must be a small and simple program which can be used as a yardstick to evaluate a system under test (a DBMS and a computer system).

DBMS benchmarks are not perfect and can be abused by vendors. Gray sites two major benchmark abuses: "Benchmark Wars" and "Benchmarketing" [14]. The "Benchmark Wars" occur between DBMS vendors trying to maintain top performance on a specific benchmark. If one vendor loses to another, the losing vendor reruns the benchmark with better "gurus." If the vendor succeeds in getting better results, the other vendor does the same thing. This can continue to the point were modifications are being made to the DBMS software specifically to make the benchmark run faster. "Benchmarketing" is where a benchmark is modified (or a new benchmark is created) to allow a specific DBMS product perform extremely well.

2.3 DBMS Benchmark Standards Organizations

The Transaction Processing Performance Council (TPC) is the only existing standards body for DBMS benchmarks. The TPC is a non-profit corporation founded in August 1988. The mission of the TPC is "to define transaction processing and database benchmarks and to disseminate objective, verifiable TPC performance data to the industry" [39:1]. The TPC was created because of the lack of agreed upon benchmarks for measuring DBMS performance. The TPC performs the following services:

- *Defines Standard Benchmarks*: The TPC has created three standard benchmarks to date: TPC-A, TPC-B, and TPC-C (which will be examined in this chapter), and has two more in the works: TPC-D (decision support domain) and TPC-E (enterprise domain).
- *Full Disclosure of Results*: All companies which claim a performance measure on a TPC benchmark must submit a detailed report to the TPC (called a full disclosure report). This report documents the benchmark's compliance with the TPC benchmark standard.
- *Quarterly Report*: The TPC publishes a quarterly report which contains summaries of all the benchmark results published that quarter.

The TPC has 41 current members which includes both DBMS software and computer hardware vendors.

2.4 Benchmarks for OLTP and Relational DBMSs

The following four benchmarks for OLTP and relational DBMSs are examined:

- TPC Benchmark A (TPC-A)
- TPC Benchmark B (TPC-B)
- TPC Benchmark C (TPC-C)
- Wisconsin Benchmark

The first three are the standard benchmarks defined by the TPC. The Wisconsin benchmark is a benchmark for complex relational queries. We provide detailed explanations of the similar TPC-A and TPC-B benchmarks. TPC-C and the Wisconsin benchmark are not covered in as much detail due to their complexity.

2.4.1 TPC-A and TPC-B. The TPC-A benchmark was developed in 1989 by the Transaction Processing Performance Council; TPC-B was developed in 1990. TPC-A and TPC-B use the same database and transaction profile, ACID[1] requirements, and costing formula. The major difference between the two benchmarks is that TPC-B allows the use of transaction generation processes to create transactions, while TPC-A requires the use of terminal emulation to create transactions. The TPC-A benchmark is a simple OLTP benchmark, while the TPC-B benchmark may be thought of as a database stress test.

The specifications for TPC-A and TPC-B state they "exercise the system components necessary to perform tasks associated with that class of on-line transaction processing (OLTP) environments emphasizing update-intensive database services" [14]. Both TPC-A and TPC-B are defined in terms of a banking application. The bank has one or more branches and each branch has multiple tellers (each with a terminal to the database). All the bank customers have an account. The final metric from the TPC-A and TPC-B benchmarks is throughput as measured in transactions per second.

2.4.1.1 Benchmark Database. The database consists of four tables (or files): Account, Branch, Teller, and History. The relationships between these tables is shown in Figure 1. Figure 1 is an entity/relationship diagram for the database.

The Account table contains the following fields:

- *Account_ID* (The key for the table)
- *Branch_ID* (The branch where the account is held)
- *Account_Balance*

The Branch table contains the following fields:

[1] Atomicity, Consistency, Isolation (or serializability), and Durability

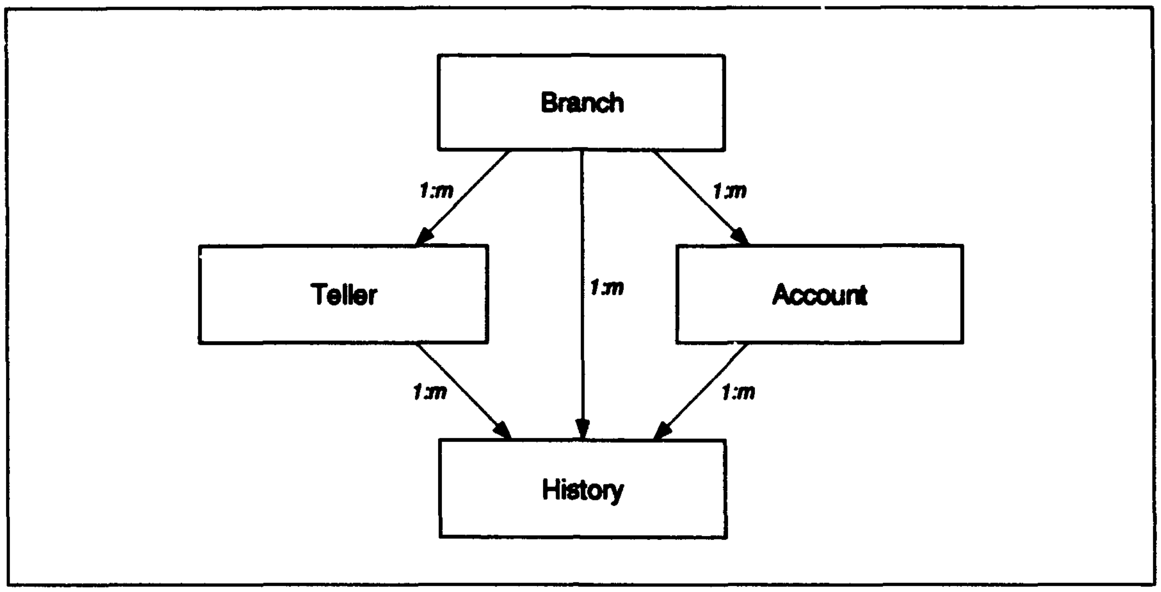

Figure 1. TPC-A and TPC-B Entity/Relationship Diagram

- *Branch_ID* (The key for the table)
- *Branch_Balance*

The Teller table contains the following fields:

- *Teller_ID* (The key for the table)
- *Branch_ID* (The branch where the teller is located)
- *Teller_Balance*

The History table contains the following fields:

- *Account_ID* (Updated by transaction)
- *Teller_ID* (Performed the transaction)
- *Branch_ID* (Associated with teller)
- *Amount*
- *Time_Stamp* (Time of the transaction)

The benchmark specification requires that all branches must have the same number of tellers and that all branches must have the same number of accounts. The number of rows

```
BEGIN TRANSACTION
  Update Account where Account_ID = Aid:
    Read Account_Balance from Account
    Set Account_Balance = Account_Balance + Delta
    Write Account_Balance to Account
  Write to History:
    Aid, Tid, Bid, Delta, Time_stamp
  Update Teller where Teller_ID = Tid:
    Set Teller_Balance = Teller_Balance + Delta
    Write Teller_Balance to Teller
  Update Branch where Branch_ID = Bid:
    Set Branch_Balance = Branch_Balance + Delta
    Write Branch_Balance to Branch
COMMIT TRANSACTION
```

Figure 2. TPC-A and TPC-B Transaction Profile

in each table is not a fixed value. It is scaled based upon the throughput rate for which the test is configured.

2.4.1.2 Benchmark Operations. Only one transaction is performed on the benchmark database. The transaction profile is shown in Figure 2. Aid (Account_ID), Tid (Teller_ID), and Bid (Branch_ID) are keys. For TPC-A, the Aid, Tid, Bid, and Delta are read from a teller terminal and the transaction is processed. Then Aid, Tid, Bid, Delta, and Account_Balance are written back to the terminal. For TPC-B, the Aid, Tid, Bid, and Delta are provided by a driver, and only Account_Balance is returned to the driver after the transaction has been processed. It is important to realize that the TPC-A benchmark measures the time it takes messages to pass through the communication network to and from the teller terminals while TPC-B does not.

2.4.1.3 Benchmark Measurements. TPC-A and TPC-B provide two important metrics: a tps and a K$/tps. The tps is a throughput measurement which stands for "transactions per second". To avoid confusion with other (older) similar benchmarks which create a tps metric (i.e., DebitCredit and TP1 [1]), the TPC-A and TPC-B benchmarks prefix the tps metrics. TPC-A uses "tpsA-Local" and "tpsA-Wide." "tpsA-Local" states the test was run using a local communications network, and "tpsA-Wide" states the test was run using a wide area communications network. TPC-B uses "tpsB" for its results.

What is required to generate a valid tps rating for TPC-A and TPC-B is not obvious and requires some explanation. The basic calculation is simple: to obtain a tps rating, the number of transactions which started and completed during the test interval is divided by the elapsed time of the test. But in order for the tps metric to be a valid for the TPC-A or TPC-B benchmark, several requirements must be met. They are as follows:

- The database table sizes must be scaled properly
- The test interval must be at least 15 minutes (and no longer than an hour)
- 90% of the transactions must have less than a 2 second response time (to the terminal for TPC-A, to the driver for TPC-B)
- Each terminal (for TPC-A) creates a new transaction (on average) every 10 seconds

First, the database table sizes must be scaled to the throughput goal of the test. TPC-A and TPC-B are scaled based upon DBMS throughput. If a benchmark test is trying to measure a throughput of 10 tps, then the database size must be scaled for that level of throughput. A tps throughput measurement is only allowed to be as high as the database table size allows. For each tps configured, the benchmark specification states that the database tables must have the following number of rows:

Table	Number of Rows
Account	100,000 rows
Teller	10 rows
Branch	1 row

In addition to the required table sizes, for TPC-A there must be 10 terminals for each tps configured.

Second, the test must be run in a steady state for at least a time of 15 minutes and no longer than one hour, but the test system must have enough resources to run the test for a total of 8 hours.

Third, 90% of all transactions during the test must have a response time of under 2 seconds.

Example: Consider a TPC-A test system configured for 10 tps using a local area network. To allow 10 tps the test database and computer system must use a minimum of:

Item	Number
Account	1,000,000 rows
Teller	100 rows
Branch	10 rows
Terminals	100

The test on the system is run for 20 minutes and the 90th percentile response time is 1.85 (below the 2 second requirement). During the 20 minutes suppose 11,261 transactions are started and committed. The tps rating would be 9.38 tpsA-Local ($\frac{11,261}{20 \cdot 60}$). Since this value is below 10 it is valid tps rating for TPC-A. But if 13,204 transactions started and committed during the test, the tps rating of 11 tps would be invalid. This is because 11 tps is larger than the 10 tps throughput for which the test system was configured.□

The second measurement from TPC-A and TPC-B is the K$/tps. This value is obtained by dividing the cost of the system by the measured tps rating. The benchmark standard is very specific about what items are to be included in the cost of the computer system. It includes cost of the computer hardware, terminals, communication lines, database software, and maintenance.

Example: In the TPC-A test above (example 1), assume that the system under test costs $140,000. The system had a throughput metric of 9.38 tpsA-Local. The K$/tps metric would be 14.9 K$/tps ($\frac{140}{9.38}$).□

These two benchmarks are widely used by DBMS vendors today. And TPC-A and TPC-B summary results are regularly published in computer industry literature.

A major strength of the TPC-A and TPC-B benchmarks is their simplicity. These benchmarks produce very simple results (tps measurements). Because of these simple results it is important to recognize the limitations of the benchmarks. TPC-A is a useful yardstick for simple OLTP performance capabilities and TPC-B provides a simple DBMS stress test, but because of the simple transaction used in both benchmarks, they are of absolutely no value for measuring how well a DBMS will perform on complex queries.

2.4.2 TPC-C. The TPC-C benchmark was developed in 1992 by the Transaction Processing Performance Council. This benchmark was designed to simulate an OLTP workload. It, like the TPC-A benchmark, is a useful yardstick for OLTP performance capabilities. The TPC-C benchmark is much more complex than the TPC-A and TPC-B benchmarks (it requires 111 pages for its specification, while TPC-A and TPC-B require 39 pages and 38 pages, respectively). TPC-C simulates a business where terminal operators execute transactions against a database. The TPC-C benchmark is specified to exercise the following components of an OLTP database system [38]:

- The simultaneous execution of multiple transaction types that span a breadth of complexity.
- On-line and deferred transaction execution modes
- Multiple on-line terminal sessions
- Moderate system and application execution time
- Significant disk input/output
- Transaction integrity (ACID properties)
- Non-uniform distribution of data access through primary and secondary keys
- Databases consisting of many tables with a wide variety of sizes, attributes, and relationships
- Contention on data access and update

2.4.2.1 Benchmark Database. The TPC-C benchmark database represents a wholesale supplier with several sales districts. The supplier has warehouses which cover a group of sales districts. Each sales district has a group of customers. For the TPC-C benchmark the following rules are specified for the benchmark database:

- Each regional warehouse covers 10 districts
- Each district serves 3,000 customers
- All warehouses maintain stocks for the 100,000 items sold by the company
- The database size is scaled by adding more warehouses (all the other cardinalities are fixed)

The benchmark database size is scaled based upon the throughput of the DBMS (like TPC-A and TPC-B).

2.4.2.2 Benchmark Operations. The TPC-C benchmark operations are based around the types of transactions which would be typical in an order-entry environment. The following transactions are run on the TPC-C database:

1. *New-Order Transaction*: This transaction enters a complete order in a single database transaction.
2. *Payment Transaction*: This transaction updates a customers balance and reflects the payment on district and warehouse sales statistics.
3. *Order-Status Transaction*: This transaction queries the status of a customer's last order.
4. *Delivery Transaction*: This transaction processes 10 new orders (the orders are delivered).
5. *Stock-Level Transaction*: This transaction determines the number of items that have a stock level below a threshold level.

All of these transactions are executed during the TPC-C benchmark. They are done in the frequency which would be expected in a real business.

2.4.2.3 Benchmark Measurements. The final metric from the TPC-C benchmark is throughput in transactions per minute. The metric is called "tpmC." As in TPC-A and TPC-B, the reported throughput may not exceed the maximum allowed by the database size.

The complexity of the TPC-C benchmark is both a strength and a weakness. It is a strength because the benchmark is more realistic (for an OLTP application), and a weakness because it makes the results of the benchmark more difficult to interpret. As with all the TPC benchmarks, the standardization of the benchmark is a strength. The benchmark leaves little flexibility in implementation (so it is less likely to be abused by vendors). A weakness of this benchmark is the single throughput (tpmC) which is generally reported (in summaries of results), but more detailed information is required in the full disclosure report.

2.4.3 The Wisconsin Benchmark. The Wisconsin Benchmark was developed by Bitton, DeWitt, and Turbyfill in 1983 [14]. This benchmark measures DBMS performance on a variety of complex relational queries. 32 queries are done on the benchmark database and each query attempts to measure DBMS performance on one, or a group of, basic relational operators (i.e., selection, projection, or join).

2.4.3.1 Benchmark Database. The benchmark database consists of three tables. The first table contains 1,000 tuples and is named ONEKTUP. The other two tables contain 10,000 tuples each and are named TENKTUP1 and TENKTUP2. The fields in the tables are all the same and are synthetically generated relations. DeWitt states that this choice was made to make the database scalable and to "permit systematic benchmarking" [14:122].

2.4.3.2 Benchmark Operations. The benchmark measures performance on the following types of queries:

1. *Selection Que ies*
2. *Join Queries*
3. *Projection Queries*
4. *Aggregate Queries*
5. *Update Queries*

There are a total of 32 queries in the benchmark specification.

2.4.3.3 Benchmark Measurements. For each of the 32 queries in the benchmark, elapsed time is used as the performance metric. This is the wall clock time from when the query was started until it completes.

This benchmark had a major impact on commercial DBMSs when it was created (1983). As DeWitt states, "by pointing out the performance warts of each system, vendors were forced to significantly improve their systems in order to remain competitive" [14:120]. For example, at the time the benchmark was released, nested loops was the only join method provided by the ORACLE and IDM 500 DBMSs [14]. DeWitt reports that "each required over five hours to execute" one of the benchmark join queries [14:134].

The Wisconsin benchmark currently is being used to evaluate the performance of database systems running on parallel processors [14].

The major strength of this benchmark is its focus on query performance. However, one has to have some education in relational database theory to understand the results. If a user doesn't understand how a selection query is different from a join query, the results from this benchmark will not be useful. But for domains, such as decision support, where complex queries are necessary, this benchmark can be helpful in evaluating the performance of a DBMS.

2.5 Benchmarks for Object-Oriented DBMSs

Though performance is important to most applications which could use object-oriented DBMSs, Cattell maintains that little work has been done in the area [8]. Only the following four benchmarks have been proposed for object-oriented DBMSs (none of which are TPC standards):

- Simple Database Operations Benchmark
- Object Operations Version 1 (OO1) Benchmark
- HyperModel Benchmark
- OO7

The Simple Database Operations benchmark was developed first. The HyperModel and OO1 benchmarks are both based on the Simple Database Operations benchmark, but HyperModel is a more complex benchmark than OO1. The OO7 benchmark is a new benchmark created at the University of Wisconsin (the creators of the Wisconsin Benchmark). Figure 3 shows the evolution of object-oriented DBMS benchmarks. Each of these benchmarks is examined next.

2.5.1 Simple Database Operations Benchmark. This benchmark was proposed in 1987 by Rubenstein, Kubicar, and Cattell [35]. They created the benchmark because existing relational DBMS benchmarks were poor measures for the applications they were working on. They needed a measure of "*response time* for simple queries" [35:387].

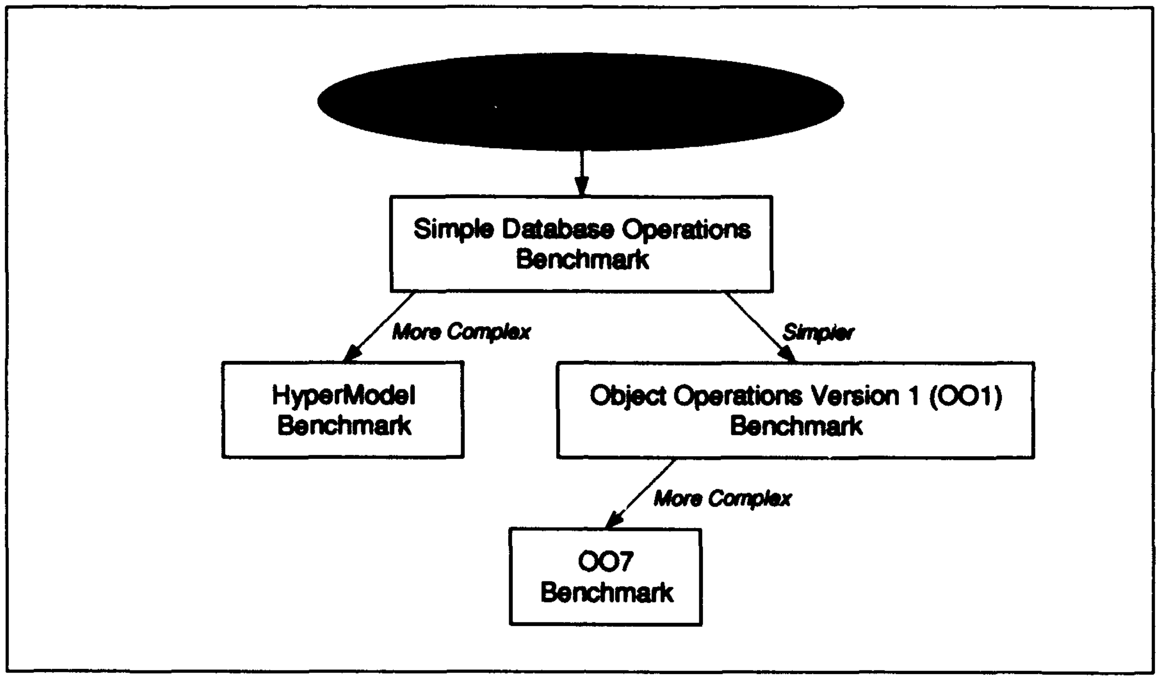
Simple Database Operations Benchmark
More Complex
Simpler
HyperModel Benchmark
Object Operations Version 1 (OO1) Benchmark
More Complex
OO7 Benchmark

Figure 3. Evolution of Object-Oriented DBMS Benchmarks

As an example, consider drawing a polygon on a computer screen where the lines which make up the polygon are stored in the DBMS. The program would start by querying the database for the first line in the polygon. When that line was returned, it would be drawn on the screen. This process would be repeated for each line in the polygon. In a complex CAD drawing there could be hundreds of thousands of lines. This is the type of application for which Rubenstein, Kubicar, and Cattell were interested in providing a benchmark, but they used a more comprehensible database of documents and authors rather than polygons for their benchmark.

2.5.1.1 Benchmark Database. The benchmark uses a database of *document* and *person* records. Documents are related to people by a relationship called *author*. 5,000 documents, 20,000 persons, and 15,000 author relationships are created in the benchmark database. To allow the benchmark to "scale up" to larger databases, the benchmark proposes the same measurements also be run on a database ten times and one hundred times larger [35:389].

2.5.1.2 Benchmark Operations. For each different size database, the performance of the following seven different operations is measured:

1. *Name Lookup*: Find the name of a single person.
2. *Range Lookup*: Find the names of people with birth dates in a particular 10-day period.
3. *Group Lookup*: Given a random document, find all authors for that document.
4. *Reference Lookup*: Find the name and birth date for a single author of a random document.
5. *Record Insert*: Create a new author record and add it to the database.
6. *Sequential Scan*: Retrieve, one at a time, the title of every document in the database.
7. *Database Open*: Perform all the operations necessary to make the DBMS available to run an application program.

2.5.1.3 Benchmark Measurements. For each of the benchmark operations the performance measurement is the response time of the operation [35]. The response time is the elapsed time from when the operation is started until it completes.

The scaling of the benchmark database in this benchmark is limited to only three sizes. This is a weakness of this benchmark and most object-oriented DBMS benchmarks. Most of these benchmarks intend to measure performance of the object-oriented DBMS when the entire database can fit in memory, and then when it cannot fit in memory. The main strength of this benchmark is that it is quite simple, but it has not turned out to be very popular. It has also been overshadowed by the OO1 benchmark.

2.5.2 HyperModel Benchmark. This benchmark was proposed in 1990 by Berre and Anderson [15:75–91]. The HyperModel benchmark is a very complex benchmark for object-oriented DBMSs because it measures a large number of different operations. The creators of the HyperModel benchmark concluded that the Simple Database Operations benchmark did not measure enough database operations on a sufficiently complex database to be representative of a wide variety of engineering applications [15].

2.5.2.1 Benchmark Database. The HyperModel benchmark uses a database which represents hypertext. Hypertext consists of nodes and links. Nodes contain information such as text, graphics, or sound. Links maintain relationships between pieces of information. Berre and Anderson state that "Hypertext has been proposed as a good model for use in Computer Aided Software Engineering (CASE) because it is possible to store software and documentation as hypertext" [15:75].

2.5.2.2 Benchmark Operations. The following operations are measured in the HyperModel benchmark:

1. *Name Lookup*: A lookup is performed on a hypertext node.
2. *Range Lookup*: A range of hypertext nodes is looked up.
3. *Group Lookup and Reference Lookup*: Same as the Simple Database Operations benchmark, but it is extended to one-to-many, many-to-many, and many-to-many with attribute relationships as well.
4. *Sequential Scan*: Same as the Simple Database Operations benchmark. The entire database is retrieved.
5. *Closure Traversal*: Starting at a random hypertext node, find all the nodes transitively reachable by a relationship. This is done for all the types of relationships in the database.
6. *Closure Operations*: The same as closure traversal, but an operation will be performed at each node found during the traversal.
7. *Editing*: This operation changes the text found at a hypertext node, then changes it back to its previous value. The operation is also done for a hypertext node with a graphics image (picture) stored in it.
8. *Create and Delete*: This operation creates a node and then deletes it.
9. *Open and Close*: Same as Simple Database Operations benchmark, but database close time is also measured.

2.5.2.3 Benchmark Measurements. The performance measurement is the elapsed time of each benchmark operation. The HyperModel benchmark specifies that each operation must be run 50 times and that the database must be shutdown and restarted before each new type of operation is started [15].

The realistic database used in the benchmark and the realism gained by the complexity of the benchmark operations are this benchmarks greatest strengths. The benchmark's complexity is also a weakness. The complexity makes the benchmark difficult to implement and the results difficult to understand. This benchmark has not proved to be very popular with object-oriented DBMS vendors.

2.5.3 Object Operations Version 1 (OO1). This benchmark was proposed in 1991 by Cattell and Skeen of Sun Microsystems [9]. It is simpler than the Simple Database Operations benchmark (on which Cattell also worked) and is much simpler than the HyperModel benchmark. Cattell and Skeen admit the benchmark is representative of a smaller group of engineering applications than the HyperModel benchmark, but state they were trying to create a "generic benchmark" [9:2–3]. This has some merit because the OO1 benchmark is much simpler to implement than the HyperModel benchmark. The OO1 benchmark is also known as the "Cattell" or "Sun" benchmark.

2.5.3.1 Benchmark Database. The database used for OO1 consists of connected parts. A connection goes from one part object to another part object, and a single part object may have several to and from connections. For each part, three connections to other parts are created. These connections must ensure "locality of reference" by connecting parts to parts which are closest to them (Part-id numbers which are numerically close are defined to be close together) [9:4–5].

OO1 measures performance on two different size databases, called small and large. The small database consists of 20,000 parts and 60,000 connections. The large database is ten times larger than the small, hence having 200,000 parts and 600,000 connections. The authors of OO1 intended that the small database would fit in the database management system's memory buffer (or working set), while the large database would not fit in the memory buffer. They state that fitting in the working set is the "most important distinction" between the small and large databases [9:5–6].

2.5.3.2 Benchmark Operations. The OO1 benchmark measures the performance of the following three operations:

1. *Lookup*: Lookup 1,000 (10,000 for the large database) parts in the database.
2. *Traversal*: Pick a single part then find all parts connected to it (directly or indirectly), up to seven levels deep.
3. *Insert*: Create 100 (1,000 for the large database) new parts with three connections per part.

During each of the measurements it is required that a null procedure (representing some work done in an application program) in a programming language be called at each step. This requirement makes the benchmark "interactive" [9:7].

The benchmark forbids any of the operations being done as a single database call, which is how a relational DBMS might perform the operations.

A unique feature of the OO1 benchmark is that it must be run remotely. This means that the database must reside on one computer (server), and the benchmark application (client) must reside on another. The two computers are connected via a network. Figure 4 shows this configuration. The authors state that a remote database configuration is "the most realistic representation of engineering and office databases" [9:5].

The three OO1 benchmark measurements are run ten times. The results of the first run are the "cold start results," and the "asymptotic best times" on the remaining runs are the "warm start results" [9:8].

The OO1 benchmark has been very popular with object-oriented DBMS vendors (it has become a de-facto standard), probably in large part due to its simplicity. This benchmark's simplicity and its attempt to measure the effectiveness of client caching are its strengths. Its poor coverage of the performance of a large number of the functional elements required in an object-oriented DBMS is its major weakness.

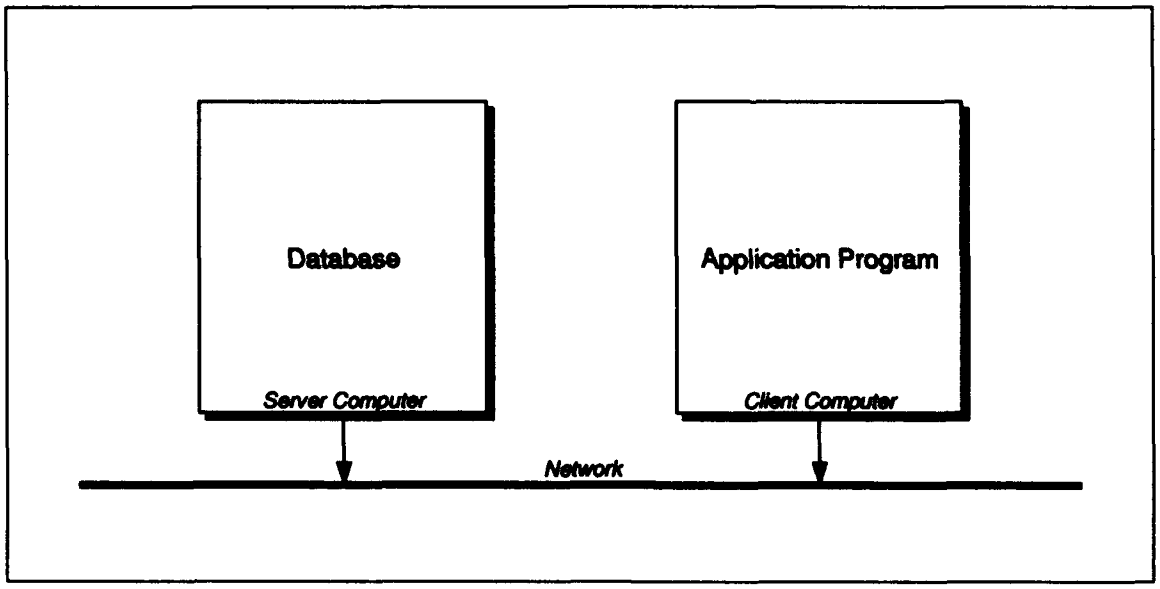

Figure 4. Remote or Client-Server Database Configuration

2.5.4 OO7. The OO7 benchmark was proposed by Carey, DeWitt, and Naughton in 1993. The work to develop this benchmark was done at the University of Wisconsin-Madison. The OO7 benchmark is proposed as a "comprehensive" performance profile of an object-oriented DBMS [7:1]. One of the interesting features of the OO7 benchmark is that it evaluates the performance of the query processor of the object-oriented DBMS (if it has one). The OO7 benchmark is more complex than the OO1 benchmark but it is more focused than the HyperModel benchmark. The OO7 benchmark produces a set of numbers as its output metrics rather than a single metric.

2.5.4.1 Benchmark Database. The benchmark database used for the OO7 benchmark is very complex. The database consists of a complex object hierarchy. The levels of the hierarchy (from bottom to top) are listed below:

1. *Atomic Parts*
2. *Composite Parts* composed of atomic parts with associated documentation objects
3. *Base Assemblies* composed of composite parts
4. *Complex Assemblies* composed of base assemblies
5. *Design Objects* composed of complex assemblies

6. *Modules* composed of design objects with associated documentation objects

Each object class in the database has several discrete attributes and connections between classes are set up in the database. The OO7 benchmark scales the database to three different sizes: small, medium, and large.

2.5.4.2 Benchmark Operations. The benchmark measures performance on the following types of operations:

1. *Traversals*
2. *Queries*
3. *Structural Modification Operations*

2.5.4.3 Benchmark Measurements. The performance measurement is the elapsed time (or response time) of each benchmark operation.

This benchmark is very new and it remains to be seen if it will become popular (specific parts of the benchmark may become popular with vendors if their product performs well on them). The benchmark is very complex and the results will probably have to be accompanied with a full description of the benchmark operations (which was done in [7]). The strength of this benchmark is that it is very comprehensive in its measurements. This benchmark measures performance on a wide spectrum of object-oriented DBMS functionality.

We have reviewed eight benchmarks which are used for the performance measurement of DBMSs. For each benchmark the database, operations, and results used by the database have been discussed. Table 1 summarizes the benchmarks covered. DBMS benchmarks can be a useful tool for evaluating commercial DBMSs, but they can also be abused. A good understanding of DBMS benchmarks can help one to know when and when not to use a benchmark.

Table 1. DBMS Benchmark Summary

Benchmark	Domain	Simplicity	TPC Standard
TPC-A	Simple OLTP	Simple	Yes
TPC-B	DBMS Stress Test	Simple	Yes
TPC-C	Complex OLTP	Complex	Yes
Wisconsin	Query Performance	Complex	No
Simple Database Operations	Object Operations	Simple	No
HyperModel	Object Operations	Complex	No
OO1	Object Operations	Simple	No
OO7	Object Operations	Complex	No

2.6 Computer Simulation

In the next two sections we examine computer simulation, including the use of object-oriented DBMSs in computer simulation systems and simulation environments. The material examined contributed to our design of the simulation benchmark.

2.7 Current Simulation Systems Using Object-Oriented DBMSs

In this section we examine the experiences of two attempts at using an object-oriented DBMS for a simulation system: one using the C++ language with an object-oriented DBMS, and one developed using the Ada language. Both of these systems previously used a relational DBMS, specifically the Oracle relational DBMS, for data management.

2.7.1 Visual Intelligence and Electronic Warfare Simulation Workbench. Woyna, et al., developed a simulation system which used the Versant object-oriented DBMS [40]. The simulation system is called the Visual Intelligence and Electronic Warfare Simulation (VIEWS) Workbench software system. VIEWS was designed to enable analysts to build detailed intelligence and electronic warfare scenarios. The scenarios created by VIEWS are used to drive high-resolution intelligence and electronic warfare models. VIEWS had been created using a relational DBMS and then modified to use the Versant object-oriented DBMS. The builders of VIEWS cited the following advantages of the object-oriented DBMS over the relational DBMS for their project:

- *Better Schema Support*: Use of the relational DBMS required that the C++ structures be "flattened" into relational tables. The object-oriented DBMS directly captured the schema from the C++ application code. Also it was noted that the translation between the relational tables to the internal C++ representation required extensive source code which was unnecessary for the object-oriented DBMS.
- *Better Application Language Interface*: Use of the relational DBMS required developer knowledge of two languages: C++ and SQL. The object-oriented DBMS did not require developer knowledge of SQL.
- *Better Application Performance*: Reconstructing the complex objects in the C++ program from the normalized relational DBMS tables required complex joins between many tables which was expensive in terms of application performance. The direct representation of C++ objects and the client cache available in the object-oriented DBMS provided improvements in performance 10 to 100 times faster than the relational DBMS provided [40].
- *Additional Features*: The object-oriented DBMS provided additional features such as long transactions and versioning of objects which were not available from the relational DBMS.

It took approximately 10 person-days of effort to convert the 30,000 lines of C++ code in the VIEWS system to the Versant object-oriented DBMS. No major porting problems were noted. The conclusion of the authors is that the use of the object-oriented DBMS in the VIEWS simulation system was a "far better approach" than using a relational DBMS [40:501].

2.7.2 Saber Wargame. Mathias extended a wargame simulation to work with an object-oriented DBMS [25]. The wargame, called Saber, was developed at the Air Force Institute of Technology and originally interfaced with flat-files and the Oracle relational DBMS. Mathias replaced the flat-file and relational DBMS interfaces with an interface to the Science Applications International Corporation's (SAIC) object-oriented DBMS. The SAIC object-oriented DBMS was developed for the US Air Force to replace an existing COBOL-based data management system.

Saber uses the SAIC object-oriented DBMS as a data repository, but simulation execution is not done persistently and no transaction model exists to allow concurrent access to executing simulation data. Mathias found the performance of the SAIC object-oriented DBMS to be slower than expected during large data transfers, and concluded that it seemed "ill-advised" to blindly replace an relational DBMS or flat-file system with an object-oriented DBMS [25].

2.8 Simulation Environments

Rooks proposes that simulation systems are composed of a three-level hierarchy [34]. Rooks' hierarchy is shown in Figure 5. Models are created in a simulation language, which is a part of a simulation system. Rooks proposes that although most simulation systems are created around a simulation language, the attention should be around the simulation system. He compares the design of a simulation system to that of a graphical user interface (GUI). The GUI may not be perfectly suited for each application it supports, but the imperfections are forgivable when compared to the amount of redundant effort saved in the development of each application. Rook proposes the following factors for evaluation of a good simulation system: *generality*, *completeness*, *suitability*, and *flexibility* [34].

Pidd states that computer simulation systems have evolved from custom simulation program creation (where the simulation program embodied the simulation logic and hard-coded the data) towards the use of data-driven simulations [32]. Data-driven simulations reduce the time and effort required to simulate a system because they do not require a traditional programming effort. Pidd's emphasis, like Rooks', is on reducing the amount of redundant effort required during simulation development. Pidd describes two different types of data-driven simulation: *general purpose* data-driven simulators and *domain specific* data-driven simulators [32]. General purpose data-driven simulation systems provide the following:

- A pre-programmed simulation model
- A model suited to a wide range of applications
- No traditional programming by the user

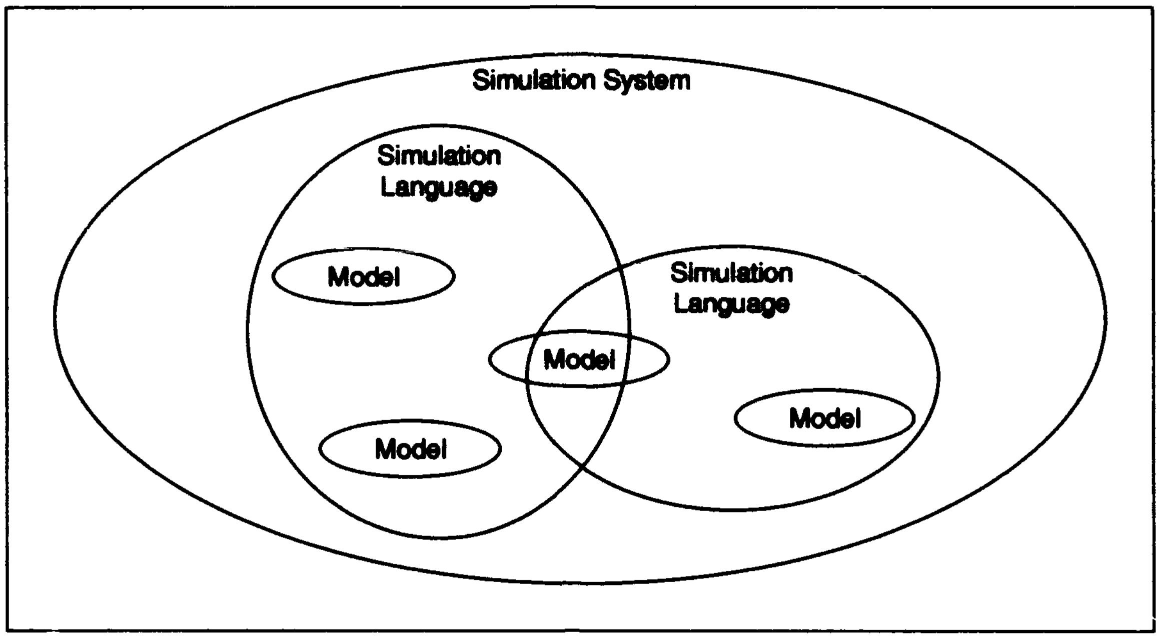

Figure 5. Three-Level Hierarchy of a Simulation System

- User provided data to the simulator
- Numerical, logical, or textual simulation data

Domain specific data-driven simulators attempt to provide the advantages of a general purpose data-driven simulation system, but are more specific to a single problem domain. A domain specific data-driven simulator must contain simulation logic for all anticipated instances of the domain. Pidd proposes that all data-driven simulations must contain the modules shown in Figure 6 [32].

2.8.1 Post-Processing and Simulation Graphics. Hurrion reports that graphical displays have been used in discrete-event simulation systems since the mid-1970s [31]. Hurrion notes that two different approaches to graphical display output have developed:

- *Post-Processing or Playback Graphics*: This technique animates the dynamics of a simulation but does not allow user interaction while the simulation is running. This technique is the most common technique used in the United States.

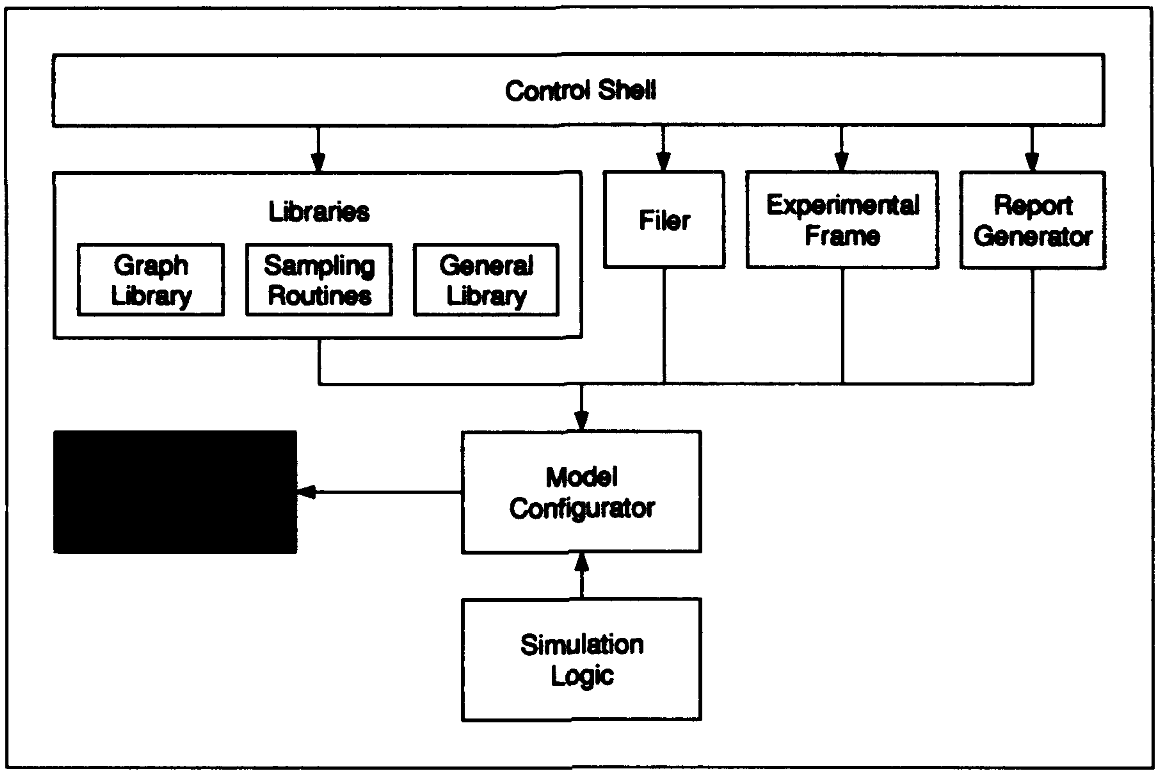

Figure 6. Components of a Data-Driven Simulator

- *Visual Interactive Simulation*: This technique is similar to the previous technique, but allows a user to interact with the running simulation. This technique is in common practice in the UK.

Hurrion also describes two different types of simulation graphics output: *character graphics* and *high-resolution bit-mapped graphics* [31]. Character graphics use repeated drawing and erasing of characters on a text screen to animate a simulation model. High-resolution bit-mapped graphics produce superior quality animation of simulation models, including three-dimensional animation. Hurrion notes that despite the visual superiority of bit-mapped graphics, a substantial amount of time is required to create a quality bit-mapped graphics animated display.

2.8.2 The Joint Modeling and Simulation System. The Joint Modeling and Simulation System (J-MASS) is being developed by the Department of Defense (DOD) as a standard architecture for modeling and simulation [6]. J-MASS provides a modeling system designed to support the simulation requirements of the DOD. J-MASS consists of two major parts: the *simulation support environment* and the *modeling library*.

The simulation support environment (SSE) is a work environment designed to assist a modeler in five functions:

- *Develop Model Components*: This portion of J-MASS allows a model developer to create individual components which are compliant with the J-MASS architecture. For example, a model developer could develop an engine component and an avionics component for an aircraft evaluation model.
- *Assemble Model Components*: This portion of the SSE allows the composition of model components, developed previously and stored in the modeling library, into new components. For example, an aircraft component could be constructed from an engine and avionics component stored in the modeling library.
- *Configure Simulation Scenarios*: This portion of J-MASS allows the developer to place a model in a simulation scenario. At this phase, specific locations and terrain may be specified for a simulation.
- *Execute Simulations*: This function of J-MASS allows execution of a simulation. The results of the simulation are journaled and made available for analysis during post-processing. The specific results to be collected for post-processing are specified by placing "probe" points in the model [6].
- *Post-Processing*: This function allows the J-MASS user to analyize the output of a single simulation run or several simulation runs.

The modeling library provides J-MASS users with a source of validated modeling components. Users of J-MASS may modify the existing components in the modeling library by changing attributes of the model components. The modeling library provides a model reuse repository for J-MASS [21].

2.8.3 J-MASS Data Management.

The current version of the J-MASS system uses flat-files for data management. J-MASS is currently being expanded to use a DBMS, but the DBMS use is limited to the model development, assembly, configuration, and post-processing functions of the system. J-MASS does not currently plan to allow execution of the simulation in a persistent environment, allowing concurrent access to executing simulation data. Current J-MASS simulations are developed, compiled, and executed as Ada programs.

2.9 Summary

In this chapter we have reviewed current literature about DBMS benchmarks and computer simulation. In Chapter III we examine the functional capabilities of the three commercial object-oriented DBMS used for this research.

III. Object-Oriented DBMS Functional Comparison

In this chapter we examine the functional capabilities of the Itasca, Matisse, and ObjectStore DBMSs. After an overview of the three DBMSs, we examine several functional areas which are thought, in current literature, to be important for an object-oriented DBMS to support [2, 4, 8, 37]. We examine transaction properties, locking and concurrency control, security authorization, query capabilities, distributed database capabilities, programming-language integration, and architecture. The topics we examine in this chapter are far from exhaustive, but catalog the areas encountered during this research effort.

3.1 DBMS Overview

We selected the Itasca, Matisse, and ObjectStore DBMSs for our research because they represent a cross-section of the commercial object-oriented DBMS industry. Table 2 shows the estimated 1993 market share of various companies in the object-oriented DBMS industry [33]. The company names of the DBMSs we are using are set in italics. ObjectStore leads the object-oriented DBMS industry with a 35% market share. Matisse holds a solid 5% market share, while Itasca (which is a young company) holds a 2% market share.

Cattell in [8] describes the ObjectStore and Itasca DBMSs as *object-oriented database programming languages*. This class of object-oriented DBMSs extend a programming language with databases capabilities. The ObjectStore DBMS extends the C++ programming language, while the Itasca DBMS extends the Lisp programming language.

The Matisse DBMS is not examined by Cattell in [8], but would roughly fall into two categories: *object manager* and *database system generator*. At the lowest level, Matisse acts as an object manager. It provides no query language, and has a very basic data model. Through the use of *data model templates*, Matisse acts as a database system generator. A data model template tailors the database to a specific data model. Creation of these templates is difficult, and certainly not automated, but provides some of the capabilities of a database system generator.

Table 2. Estimated 1993 Market Share of Object-Oriented DBMS Companies [33]

Company	Annual Revenue (in millions)	Market Share
Object Design	$20.4	35%
Servio	$7.3	13%
Objectivity	$6.6	11%
Versant	$5.9	10%
Ontos	$3.0	5%
Intellitic	$2.6	5%
BKS	$2.7	5%
O2 Technology	$1.9	3%
HP	$1.8	3%
Itasca	$1.2	2%
DEC	$1.1	2%
UniSQL	$1.1	2%
Other		4%

3.2 Transaction Properties

The three commercial object-oriented DBMSs examined in this research support the concept of a transaction. This section examines the support the three databases provide for ACID transactions, long transactions, nested transactions, and nonblocking read-only transactions.

3.2.1 ACID Requirements. Itasca, Matisse, and ObjectStore support transactions which pass the ACID test: ***atomicity***, ***consistency***, ***isolation***, and ***durability***. The ACID requirements for transactions are the accepted norm for all of today's relational DBMSs. The TPC benchmarks require that all transactions meet the ACID test.

3.2.2 Long Transactions. Support for long transactions differs between the three databases. Itasca and ObjectStore provide support for long transactions via a version system. Versioned objects are checked out into work areas. Matisse does not provide support for long transactions.

3.2.2.1 Itasca. The Itasca DBMS supports long transactions through version management. Objects are worked on in a work area called a ***private database***. When a user wants to work with a group of objects for a long period of time, they are checked out of the Itasca *shared database* into the user's private database. This checkout creates a new version of all the objects checked out in the user's private database. In the current Itasca implementation, the new version is a copy of all the objects, not a delta, so a substantial amount of disk space may be required for this operation. After check out, the objects which remain in the shared database become read only. When the user is finished with work, the objects in the private database are checked back into the shared database, and are now available to other users.

3.2.2.2 ObjectStore. The ObjectStore DBMS also supports long transactions through version management. To work on a group of objects for a long period of time in ObjectStore, the objects are grouped into a *configuration* and checked out into a *workspace*. The check out of the configuration creates a private version of all the objects in the configuration. When the user is finished with work on the configuration it is returned, or checked in, to the workspace from which it came.

ObjectStore allows more than one user to check out a version of a configuration. If several different versions of the configuration have been created, then the versions must be merged. ObjectStore provides support for version merging.

In an ObjectStore database, every object inside a configuration has a *version history* which describes all the versions of an object which have been created. The past versions of the version history are able to be read, but cannot be changed. The configuration groups objects together as a unit for versioning. A configuration is also the unit of concurrency control, so granularity for locking must be taken into account in the design of configurations. When a new version of a configuration is created, a new version of all the objects in the configuration is created. ObjectStore minimizes the storage overhead required for this operation by only storing the differences, or deltas, between database pages which make up the configuration [28].

ObjectStore provides workspaces to allow organization of work, and to allow work on a distinct version of a configuration. The workspaces in ObjectStore form a parent/child hierarchy of arbitrary depth. The top node of the tree is called the *global workspace*. The workspace hierarchy is created based upon the needs of the workgroup which it is designed to support.

3.2.2.3 Matisse. The Matisse object-oriented services application programming interface (API) does not provide support for long transactions [19]. Matisse provides a version management system but it is limited to creating object versions in a linear manner. New versions of an object are always created from the most recent version and no "forking" of the version tree is allowed. Matisse support informed us that this capability will be available in version 2.3 of the Matisse DBMS.

3.2.3 Nested Transactions. Nested transactions are useful in an application where a single session may contain several individual changes to data. Each individual change may be committed or aborted and then the entire session may be committed or aborted. An abort of the session would abort all the individual changes [8]. Only ObjectStore and Itasca support nested transactions at this time.

The ObjectStore DBMS provides support for nested transactions. In ObjectStore, transactions may be freely nested. The inner transactions depend upon the outer transaction to determine if they are committed. The Itasca DBMS provides support for nested transactions through database sessions. An Itasca database session is a sequence of transactions. Itasca sessions may be created and destroyed in a "stack-like" manner [13:39]. The Matisse object-oriented services API does not provide support for nested transactions.

3.2.4 Non-Blocking Read-Only Transactions. The Matisse DBMS provides a unique feature due to its use of *intrinsic versioning*. The Matisse DBMS allows an application to perform an "as of" transaction. The transaction is a *non-blocking read-only transaction* which reads the state of the database at a specific logical database time. The transaction reads the version of the object "as of" the logical database time specified, and since (under intrinsic versioning) a new version of the object will be created if another transaction changes the object, no locking is required.

ObjectStore and Itasca also allow read-only access to previous versions of objects. But the application must have created, and be responsible for managing those versions. New versions are not created every time an object is changed, as in Matisse.

3.3 Locking and Concurrency Control

Concurrency control in Matisse and Itasca is at the object level; in ObjectStore concurrency is at the database page level. If configurations are being used in ObjectStore, then concurrency is at the configuration level. The differences between page level and object level concurrency will be examined further in Section 3.8.

3.4 Security Authorization

Authorization to use a database file in ObjectStore and Matisse is based upon UNIX file system security. If a user has access to a file through UNIX, then that user has access to it through ObjectStore or Matisse. Essentially, these databases rely upon the operating system to provide security.

The Itasca database has a much more sophisticated security authorization system. Every user of the Itasca database has an identifier, which is independent of the operating system user identifier. The identifier is used to control access to objects in the Itasca shared database and to provide access to various private databases. Every operation on an object in Itasca is checked to ensure that the user has authorization. This security checking incurs a large amount of overhead. So Itasca does not do access checks on each object when a user works inside a private database. In other words, if a user has access to a private database, the user is granted access to all the objects in the private database.

3.5 Query Capability

All three of the DBMSs provide an "ad-hoc" query capability through a graphical browser tool, but only ObjectStore and Itasca provide query language support from within a database application. The Matisse DBMS does not provide a query language capability

```
ItascaSet obj_list;

Vehicle::select(
  obj_list,
  QUERY_EXPRESSION, "( equal ManufacturerName \"Oldsmobile\" )",
  END_ARGS );
```

Figure 7. A Sample Itasca Query

in the object-oriented services API. Itasca and ObjectStore differ in the implementation of their application query languages.

3.5.1 Itasca. Itasca provides a query capability for DBMS applications. The query language is limited to performing queries over instances of classes [3, 20]. The scope of a query may be set by the user of the database to *private*, *shared*, or *global*. If the scope of a query is private, only instances of the class in the current private database are examined during the query. If the scope of the query is shared, then only instances of the class in the shared database are examined during the query. If the scope of the query is global, then all instances of the class in the current private database and the shared database are considered. An example of an Itasca query in the C++ API is shown in Figure 7. The query places all instances of the *Vehicle* class which have a value of "Oldsmobile" for their *ManufacturerName* into the *obj_list* set. Notice that the query expression must be written in the Lisp programming language, not the C++ language. This is a disadvantage of the current Itasca C++ API because the programmer must understand the C++ and Lisp programming languages along with the additional semantics of the Itasca DBMS.

Itasca also provides a *query optimizer* to optimize the evaluation of application queries.

3.5.2 ObjectStore. ObjectStore also provides a query language capability which allows complex queries on collections. Collections in ObjectStore are groups of objects. An example of a query in ObjectStore is shown in Figure 8 [28]. The query determines all the public school students who are also teenagers. The query shown in Figure 8 uses the ObjectStore DML. The ObjectStore DML extends the C++ language with database

```
os_database *student_database;
os_Set<student*> &public_school_students;

os_Set<student*> &teenagers =
  public_school_students[: this->age >= 13 && this->age <= 19 :];
```

Figure 8. A Sample ObjectStore DML Query

constructs. The DML query language is a natural extension of the C++ programming language. An ObjectStore query may also use the *query* method, which is defined for every ObjectStore collection class.

ObjectStore provides a *query optimizer* to optimize the evaluation of application queries.

3.6 Distributed Database Capabilities

All three DBMSs provide client/server type distributed capabilities, but only the Itasca DBMS manages fully distributed databases. To set up a distributed database in Itasca, several databases on different machines are initialized then declared to be *network sites*. Data can migrated among network sites through the use of database administration tools provided by Itasca. The Itasca DBMS's distributed capabilities are very similar to the capabilities provided by existing distributed relational DBMSs.

3.7 Programming-Language Integration

This functional criteria was identified by Cattell [8]. Cattell maintains that an object-oriented DBMS should be closely integrated with a programming language to solve the *impedance mismatch* problem which is common in relational DBMSs. If a close relationship exists between the programming language and the database, a programmer will only have to learn how to use one programming language.

3.7.1 Itasca. The Itasca DBMS has close ties with the Lisp programming language, but Itasca has created an interface to the C++ programming language. For this research we only investigated the C++ interface to Itasca. So we are unable to examine how well the Itasca DBMS is integrated with the Lisp programming language. The main design goal of Itasca when they developed the C++ API was to provide the full capabilities of the existing Itasca database through the C++ programming language. Therefore, the C++ API is closely tied to the C++ language where the C++ object model and the Itasca object model are the same. But where they are different the C++ API forces the Itasca DBMS model upon the C++ programmer.

3.7.2 Matisse. The Matisse DBMS makes no attempt to provide close ties with a programming language. The interface to Matisse, which is defined in the C programming language, is a functional interface into the database's object model. If a programming language (such as C++) supports an object model, then the programmer must learn the language's object model and the Matisse DBMS's object model. This separation ensures that Matisse is not closely tied to any single programming language.

3.7.3 ObjectStore. The ObjectStore DBMS provides close integration to the C++ programming language. ObjectStore defines a superset of the C++ programming language, called the ObjectStore DML, which makes database functionality available to the programmer. ObjectStore also provides a more functional interface to standard C++ (C++ without ObjectStore's language extensions) and the C programming language.

3.8 Database Architecture

This section will examine some of the available information on the architectures of the three commercial DBMSs examined for this research and the major differences we observed between the three DBMSs during this research.

A major difference between ObjectStore, Itasca, and Matisse is the way data is passed from the DBMS server to an application. ObjectStore is a *page* server rather than an *object* server. The Itasca and Matisse DBMSs provide data to a client application as objects. For example, if a single object is requested, that single object can be sent to the application. ObjectStore transfers data to a client application in database pages. A database page in ObjectStore could contain several objects. So if an object is transferred to an application, several other objects will also sent. The idea of the page server is to improve performance. If an application works in a small area of its database, fewer transfers will be required sending database pages then sending objects. The disadvantage of the page server is that it is difficult to obtain fine grain concurrency. The ObjectStore DBMS locks data elements at the page level rather than the object level. The server has to lock all the objects on a page, when an application may only be working with a single object.

ObjectStore executes an ObjectStore server process, *osserver*, on any machine which acts as a database server. Every machine which will be a client machine must be executing a cache manager process, called *cmgr*. Only one cache manager is executed on a client machine, even if multiple clients are executed. The requirement for the cache manager process is a very annoying part of ObjectStore. Neither Matisse nor Itasca required any software on the client machines, except the client application (although Matisse required access to the license manager). The cache manager could also become a bottleneck if several applications were executed on the same client. However, this is unlikely since a single workstation usually only serves one user.

3.8.1 Itasca.

Very little is discussed about the architecture of the Itasca DBMS in the documentation provided with the product, but the architecture of the Orion DBMS is described in [13, 22, 23]. The Itasca DBMS is the closest to a traditional relational DBMS architecture of the three object-oriented DBMSs we examined. It provides a database server which is responsible for executing all interactions with the DBMS. Some support for caching of data is allowed in Itasca, but we encountered problems when using it in the C++ API. Itasca support is creating better support for caching in a new version of the C++ API, which should improve the performance of remote applications.

3.8.2 Matisse.

The architecture of the Matisse database is composed of three different levels built upon each other [19]:

- *Micro-Model*: The micro-model is the lowest level of the Matisse database. It implements a very simple data model of objects and connections. The micro-model is used to build templates.

Table 3. Summary of Object-Oriented DBMS Capabilities

Capability	Itasca	Matisse	ObjectStore
ACID Transactions	Yes	Yes	Yes
Long Transactions	Yes	No	Yes
Nested Transactions	Yes	No	Yes
Non-Blocking Read-Only Transactions	No	Yes	No
Locking Level	Object	Object	Page or Configuration
Security	Database	Operating System	Operating System
Application Query Language	Yes	No	Yes
Distributed Database	Yes	No	No
Close Language Interface	Yes (Lisp)	No	Yes (C++)
Server Type	Object	Object	Page

- *Templates*: A template is a generic data model which describes the rules to follow when designing a database schema. The only template which exists for Matisse is an object-oriented template. This template is used by the object-oriented services API. Matisse states that any data model template can be developed in on top of the micro-model. For example, a relational data model could be developed for Matisse.
- *Schema*: A schema is a user-defined structure which is used by an application.

3.8.3 ObjectStore. The ObjectStore architecture is designed to enable a client application very fast access to data. ObjectStore uses *virtual memory mapping* and *client caching* to improve application performance. Virtual memory mapping allows persistent objects to be mapped via normal memory locations. If a program tries to access an object which has not been moved to memory, a fault occurs, and the object is brought into memory. This process is transparent to the application programmer. The advantage of the ObjectStore approach is that access to persistent data once it has been mapped into the application's virtual memory is as fast as normal memory access [28]. However, ObjectStore's database size is limited by the virtual address space of the machine, which is 2^{32} bytes on a Sun SPARC machine.

3.9 Summary

This chapter has examined the functional capabilities and differences between the Itasca, Matisse, and ObjectStore DBMSs. Several topics of database functionality important to this research were investigated. A summary of the functional capabilities of Itasca, Matisse, and ObjectStore is shown in Figure 3. The next two chapters examine our work with the OO1 benchmark and the simulation benchmark.

IV. OO1 Benchmark Analysis, Design, and Implementation

In this chapter we describe our analysis, design, and implementation of the OO1 benchmark for the Itasca, Matisse, and ObjectStore object-oriented DBMSs. Object-oriented analysis and design was used on the benchmark requirements to produce a valid design. Our analysis and design used the methods presented by Coad and Yourdon [10, 11] and by Rumbaugh, et al. [36]. Implementation of the benchmark was done on the three commercial object-oriented DBMSs based upon the final design. This chapter first examines our object-oriented analysis and design of the OO1 benchmark and then examines the issues faced when implementing the benchmark on the three object-oriented DBMSs.

4.1 Object-Oriented Analysis

Our analysis was based upon the information contained in the benchmark specification [9, 14]. The specification defines requirements for the OO1 benchmark but does not dictate a specific implementation. The goal of our analysis was to build an object-oriented model of the system required by the benchmark specification. To do this, the steps of the Coad/Yourdon object-oriented analysis (OOA) method presented in [10] were used. These steps are listed below:

1. Identify classes and objects
2. Identify structures
3. Identify subjects
4. Define attributes
5. Define services

Our final Coad/Yourdon OOA diagram is presented in Figure 9. A summary of Coad-/Yourdon notation appears in Appendix A. How we arrived at this diagram will be explained, in detail, in the following sections which describe the steps taken in our analysis.

4.1.1 Identify Classes and Objects. The OO1 benchmark centers around a single object: the part. Various actions are performed on parts and the wall clock time, or elapsed

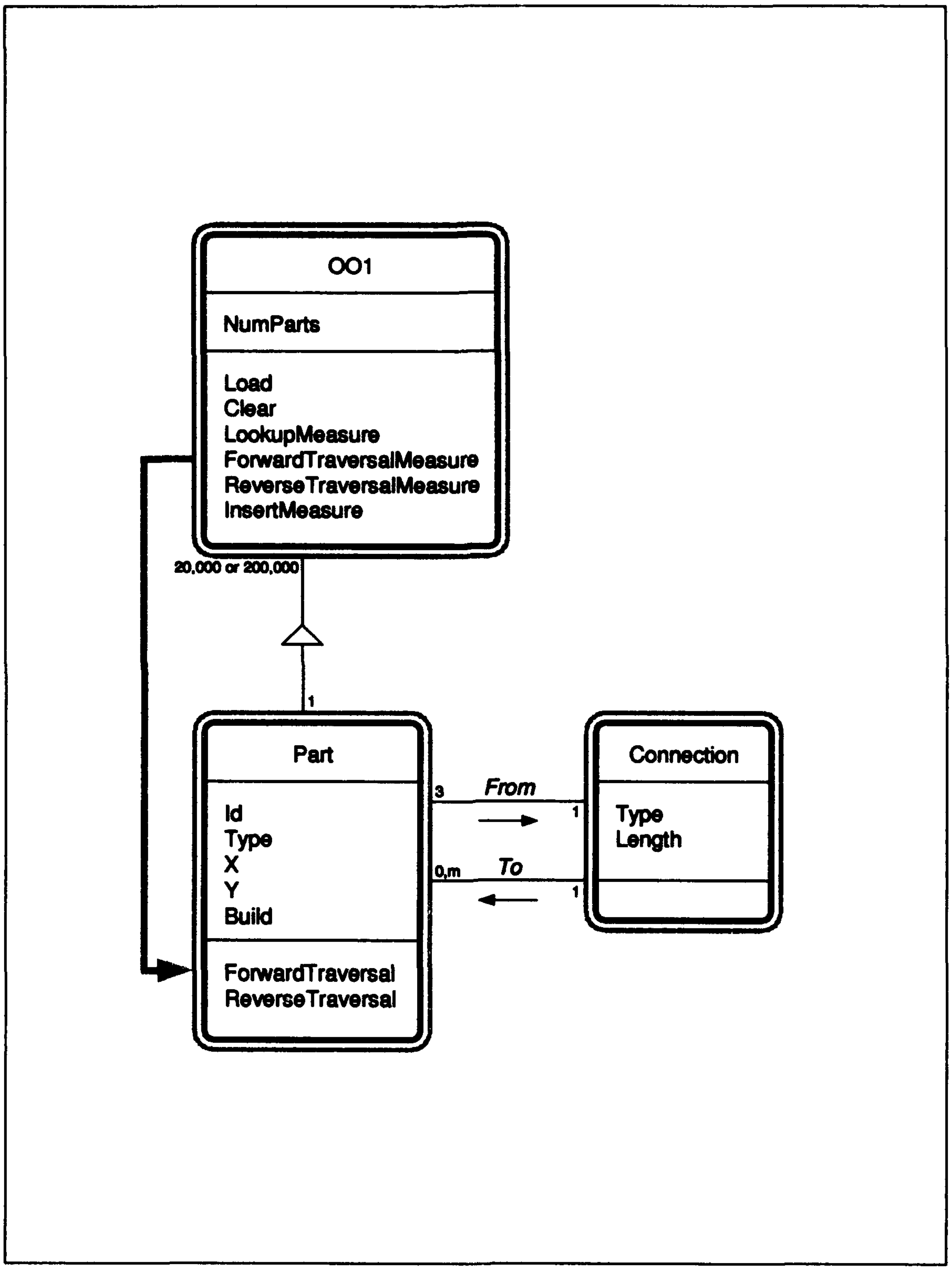

Figure 9. OOA Diagram for the OO1 Benchmark

time, required to perform them is measured. Therefore, the first class in our analysis model was the *part* class. The benchmark requires that a group of parts be created in the database (20,000 for the small database, and 200,000 for the large database). Each part connects to three other parts and must know which parts connect to it. Each part contains, as attributes, some information about itself. Each connection also contains some information about the connection it represents. Figure 10 shows a group of five parts and some of their connections. The connections are only shown for the part with an identifier value of 1, and the part with an identifier value of 4 (part 1 and part 4). All of the other parts would have three connections also, but these connections are not shown in Figure 10. Each part knows to which parts it connects. For example, part 1 knows that it connects to part 1, part 2, and part 2 (again). Also, part 4 knows that it connects to part 1, part 2, and part 3. The benchmark specification does not mandate that the connections be kept in any particular order. Each part also knows what parts connect to it. For example, part 1 knows that part 1, and part 4 connect to it. Hence, the arrows in Figure 10 can be followed in the reverse direction, as well as in the forward direction. Notice that the number of connections from a part is fixed at three, but the number of connections to a part may vary. For example, part 1 connects to three parts, but has only two connections to it.

The second class we identified in our analysis was the *connection* class. The connection class holds the attributes required for each connection. This class is really a relationship between two parts which has attributes. In the Coad/Yourdon OOA notation, this is represented as a class, but is modeled by Rumbaugh, et al. more directly. Figure 11 shows the representation of the part and connection classes in a Coad/Yourdon OOA diagram and Figure 12 shows the same diagram represented as a Rumbaugh object model. The Coad/Yourdon OOA diagram uses instance connections to model the structure shown more directly in the Rumbaugh object model.

The last class we identified in our analysis of the OO1 benchmark was the *oo1* class. This class was needed for two reasons. First, it acts as a container for the parts and indirectly the connections. Second, it encapsulates the OO1 benchmark measures.

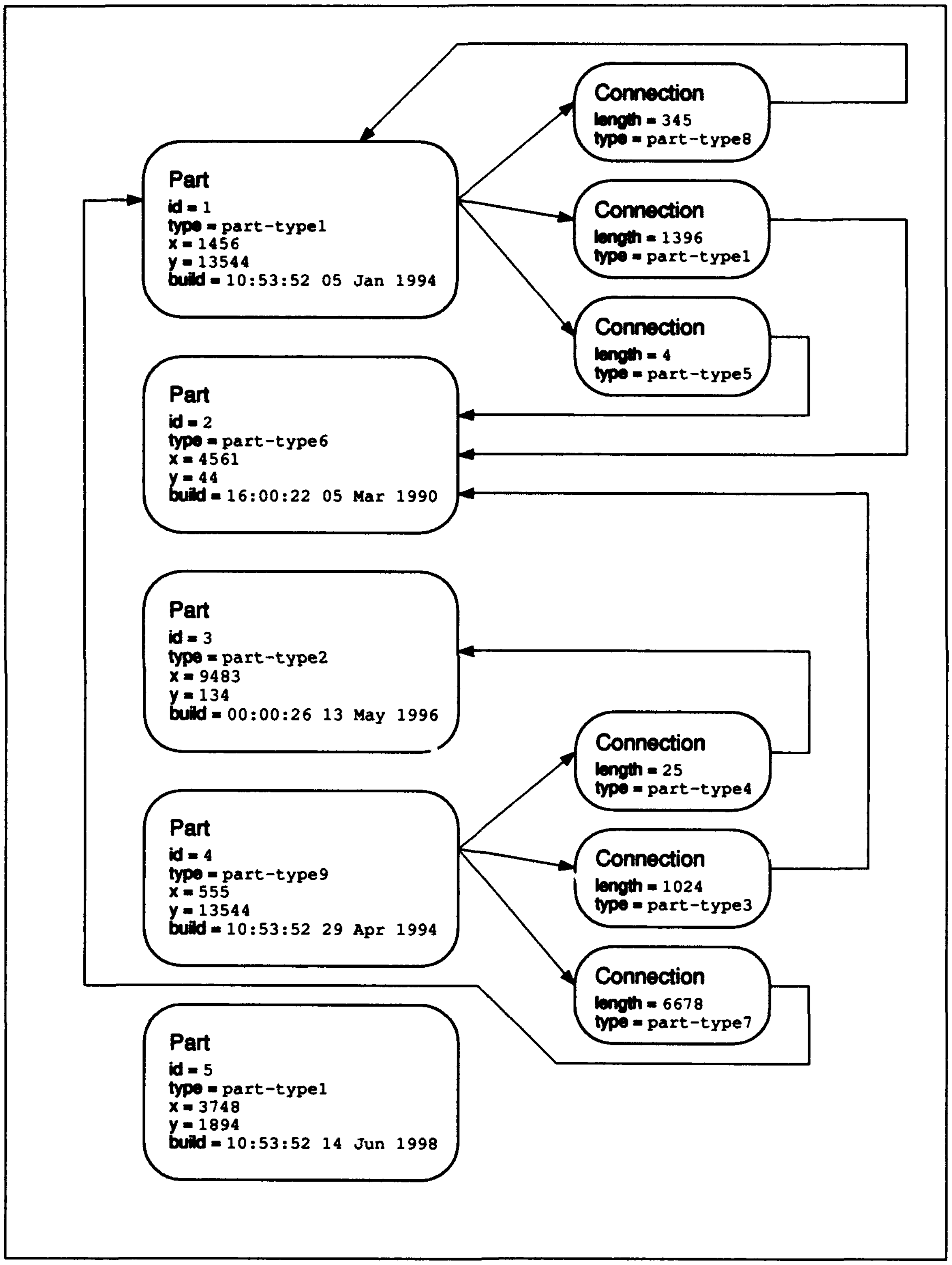

Figure 10. Parts and Connections in the OO1 Benchmark Database

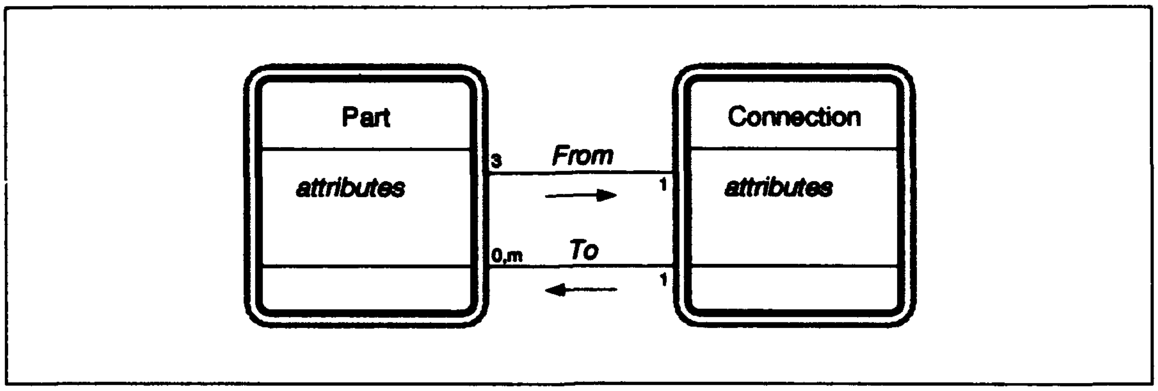

Figure 11. Part and Connection Representation in a Coad/Yourdon OOA Diagram

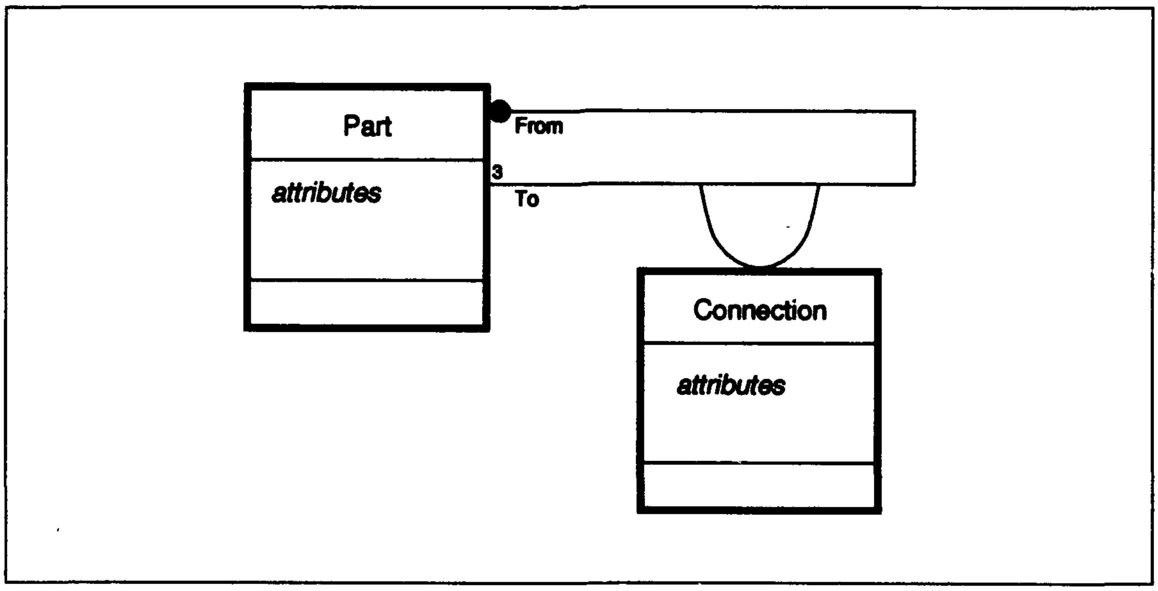

Figure 12. Part and Connection Representation in a Rumbaugh Object Model

4.1.2 Identify Structures. Coad/Yourdon specify two types of structures which should be identified during analysis: generalization-specification and whole-part [10]. Generalization-specification identifies "is a" relationships, commonly called inheritance, between classes. Whole-part structures identify "has a", or "is made up of", relationships between instances of classes. For the OO1 benchmark, we identified no generalization-specification structures, but set up a whole-part structure between the oo1 class and the part class. In our model, all part instances are contained in a single instance of the oo1 class. This whole-part structure can be seen in Figure 9.

4.1.3 Identify Subjects. Subjects are used to break an analysis model into different parts which form a logical group of classes [10]. Due to the small number of classes we identified, we omitted the identification of subjects, but subjects are used in our design for the OO1 benchmark.

4.1.4 Define Attributes. Attributes are data, sometimes called state information, for which a separate copy is included in each object of a class [10]. Most of the attributes we identified for the OO1 benchmark are dictated by the benchmark specification. The specification describes the data which must be associated with each part and connection. This simplified the definition of attributes for our analysis model. For the part class, the following attributes were identified directly from the benchmark specification:

- *Id*: This attribute acts as a unique identifier for a part, which is defined as an integer. Starting from a value of 1, each part instance is assigned a consecutive integer for its *Id* value.
- *Type*: This attribute is defined as a string. The string may take on values between `part-type0` and `part-type9`. The specific value for a part instance is randomly selected.
- *X*: This attribute is defined as an integer between 0 and 99,999. The value represents the y-axis value of a location. The specific value for a part instance is randomly selected.

- *Y*: This attribute is defined as an integer between 0 and 99,999. The value represents the y-axis value of a location. The specific value for a part instance is randomly selected.
- *Build*: This attribute represents a date and time. The value for a specific part is randomly selected from a 10-year range.

For the connection class, the following attributes were identified directly from the benchmark specification:

- *Type*: This attribute is defined the same as the *Type* attribute of the part class.
- *Length*: This attribute is defined the same as the *X* and *Y* attributes of the part class. The value represents the length of the connection between the two parts.

For the oc_ class, we identified only a single attribute:

- *NumParts*: This integer attribute contains the number of parts which have been created in the database. Its value will be 20,000 for the small database, and 200,000 for the large database.

4.1.5 Define Services. Services are the methods or functions which classes can perform [10]. As was the case for defining attributes, the benchmark specification dictated most of the required services. The services included the benchmark measures and the creation and deletion of the benchmark database. For the ool class, the following services were identified:

- *Load*: This service creates the benchmark database. For the small database, 20,000 parts and 60,000 connections are created. For the large database, 200,000 parts and 600,000 connections are created.
- *Clear*: This service removes all the persistent part and connection instances from the database.
- *Lookup Measure*: This service looks up 1,000 randomly selected parts from the database. For each part, a *null* procedure (a procedure which performs no purpose

except to exist) is called passing the X, Y, and *Type* attributes of the part[1]. The 1,000 lookups are repeated ten times. The first time is reported as the cold time for the measure and the asymptotic best time is reported as the warm time for the measure (or the average of the second through the tenth time).

- *Forward Traversal Measure*: This service finds all the parts connected to a randomly selected part, up to seven levels deep. 3,280 parts will be found in this measure. For each part, a *null* procedure is called passing the X, Y, and *Type* attributes of the part. The traversal is repeated ten times. The first time is reported as the cold time for the measure and the asymptotic best time is reported as the warm time for the measure (or the average of the second through the tenth time).
- *Reverse Traversal Measure*: This service is the same as the forward traversal, except all the parts which connect to a randomly selected part are found. An indeterminable number of parts will be found in this measure. The time for this measurement is normalized for comparison with the forward traversal measure. The traversal is repeated ten times. The first time is reported as the cold time for the measure and the asymptotic best time is reported as the warm time for the measure (or the average of the second through the tenth time).
- *Insert Measure*: This service will create 100 new parts in the database. Three new connections will also be created for each new part. As each part is being created, a *null* procedure is called to obtain values for the X and Y attributes of the part. The 100 inserts are repeated ten times. The first time is reported as the cold time for the measure and the asymptotic best time is reported as the warm time for the measure (or the average of the second through the tenth time).

For the part class, the following services were identified:

- *Forward Traversal*: This service calls the *null* procedure defined in the forward traversal measure of the oo1 class, then calls the forward traversal service for each part which is connected to it. Hence, this service is recursive.

[1]When implementing the *null* procedure, care must be taken to ensure that the compiler does not remove the call during optimization.

- *Reverse Traversal*: This service calls the *null* procedure defined in the reverse traversal measure of the oo1 class, then calls the reverse traversal service for each part which connects to it. Hence, this service is also recursive.

No services were identified for the connection class. We also note, via the shaded arrow in Figure 9, that the oo1 class calls the services defined in the part class. Specifically, the traversal measures are set up as recursive calls to the traversal services in the part class.

Our analysis of the OO1 benchmark is now complete. Due to the use of object-oriented design and then implementation in an object-oriented programming language using an object-oriented DBMS, no major changes to the model defined in our analysis are necessary during design and implementation. Our design is described in the next section.

4.2 Object-Oriented Design

The object-oriented design for the OO1 benchmark took the analysis model developed in the previous section and created a design for the benchmark. To do this, the steps of the Coad/Yourdon object-oriented design (OOD) method were used [11]. The steps to the OOD method are the same as those we used in our analysis, but attention focuses on breaking up the analysis model into the following four components:

1. Human interaction component
2. Problem domain component
3. Task management component
4. Data management component

For the OO1 benchmark, we ignored the design of the task management component, and the data management component. The task management component was ignored because the benchmark does not need to concurrently execute several tasks, and the data management component was ignored because the object-oriented DBMS will provide all of the benchmark's data management. The next two sections will describe our design for the human interaction component and the problem domain component.

4.2.1 Design of the Human Interaction Component. The OO1 benchmark needs a minimal user interface. The benchmark is run from the UNIX command line. The executable program for the benchmark is called *bench* and it accepts the following inputs:

- *Object-Oriented DBMS Authorization*: This information, which is different for each object-oriented DBMS, provides the benchmark with the information necessary to connect to the object-oriented DBMS and perform any authentication required by the security features of the object-oriented DBMS.
- *Benchmark Operation to Execute*: This information directs the program which service of the oo1 class to run. The choices include: `load`, `clear`, `lookup`, `ftrav`, `rtrav`, and `insert`.
- *Number of Parts in the Database*: To avoid a database query to determine the number of parts in the database, which could bias the cold results, the number of parts in the database is provided to the benchmark program.
- *Random Stream Number*: The random number generator used for the benchmark program contains 100 pre-defined streams of random numbers. This value selects the stream used for the measure to be executed.

The *bench* program provides text-based output which reports the benchmark results. For each benchmark measure the results include the elapsed time for each of the ten iterations and the cold and warm times.

4.2.2 Design of the Problem Domain Component. For the OO1 benchmark the problem domain component is derived from our analysis model. The following changes were made to our analysis model during design of the problem domain component:

- *Variable Number of Connections From a Part*: Our analysis model identified that each part would connect to exactly three other parts. To allow for possible variations in the number of parts a part can connect to, this was made a variable relationship. This additional capability allows for variations to be made in the benchmark database.

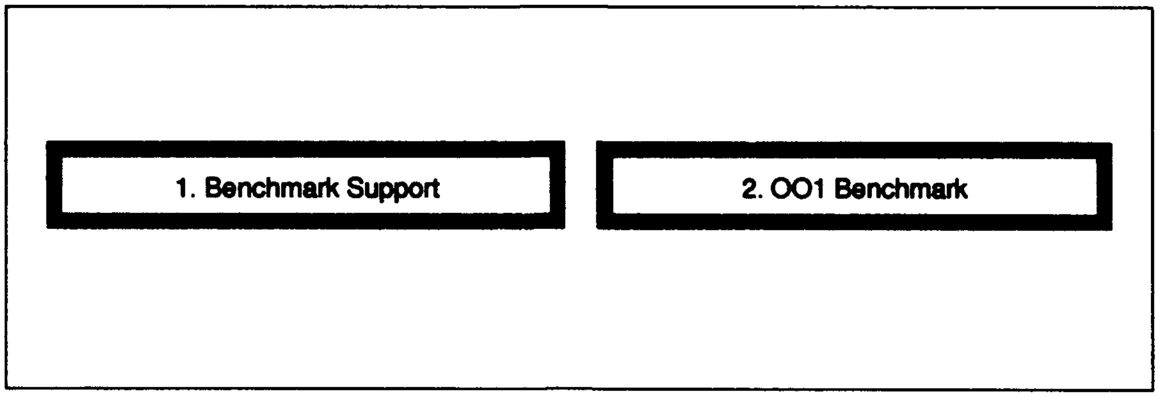

Figure 13. OOD Diagram for the OO1 Benchmark—Subject Layer

- *The OO1 Class Maintains a Collection of the Connection Instances*: To facilitate testing and to help eliminate object leaks, the oo1 class was designated to keep track of all the connection instances as well as all the part instances.
- *Identification of Persistent Classes*: The part and connection classes were identified as persistent classes. That is, they are the classes which were stored in the object-oriented DBMS. To differentiate persistent classes from non-persistent classes, the persistent classes are shaded in our OOD diagrams.
- *Addition of the Benchmark Support Library*: The benchmark support library was added as a separate subject. The benchmark support library consists of routines for measuring elapsed time and generating random numbers. This library is documented in Appendix F. The *dice* class is used by the oo1 benchmark to generate uniform random numbers and the *stopwatch* class is used to measure the duration of the OO1 benchmark operations.

Figure 13 shows the subject layer of our OO1 benchmark design and Figure 14 shows our final Coad/Yourdon OOD diagram.

Due to the use of the C++ programming language with an object-oriented DBMS, our design changed little for each of the three implementations. The next section describes our implementations of the OO1 benchmark on three commercial object-oriented DBMSs.

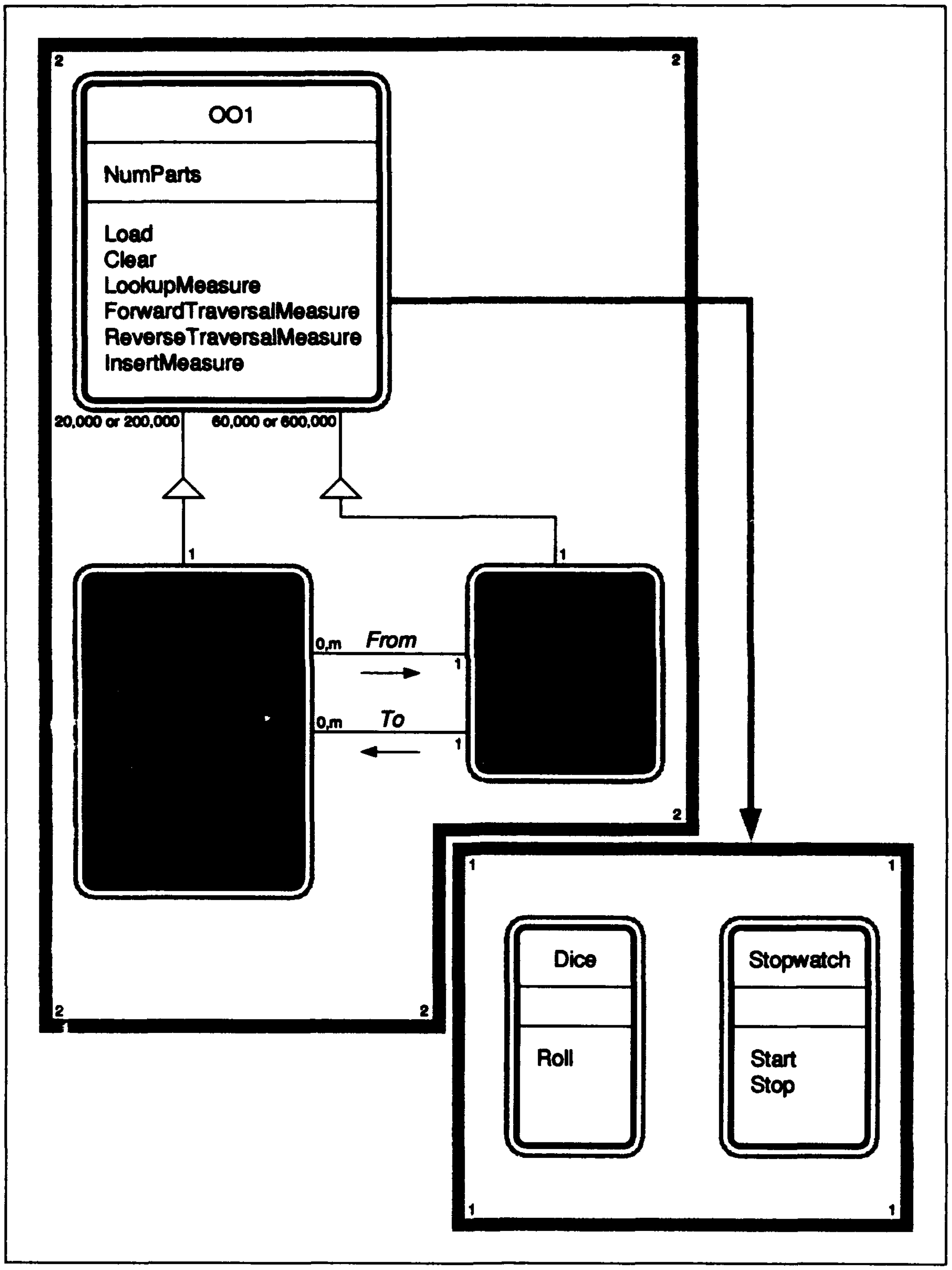

Figure 14. OOD Diagram for the OO1 Benchmark

4.3 Benchmark Implementations

This section describes the implementation of our benchmark design on each of the the three commercial object-oriented DBMSs. We first examine the changes necessary to our design to implement the benchmark on the database. Then we examine the problems we encountered implementing and running the benchmark on the database.

4.3.1 Itasca Implementation.

Our OO1 benchmark implementation for the Itasca DBMS was created using the Itasca C++ API and written in the C++ programming language. Four versions of the benchmark software were written for Itasca. We first examine the benchmark implementation in Itasca and then cover the problems encountered during implementation.

The Itasca DBMS requires that all persistent classes be subclasses of a class called *class*. Our design was changed to reflect this requirement. A final Coad/Yourdon OOD diagram for our Itasca implementation is shown in Figure 15. The *Oo1AbstractConnection* class will be discussed during the coverage of our implementation problems with Itasca.

To create the persistent classes in Itasca the *dynamic schema editor* was used. This program is the recommended method of creating classes in the Itasca database [20]. Once the classes were created, the C++ definitions were dumped into a C++ header file (this file was called *schema.hh*). The entire benchmark was implemented with a single C++ executable program, called *bench*.

Because the Itasca DBMS was implemented in the Lisp programming language, we considered creating a Lisp version of the OO1 benchmark for Itasca. But Itasca support told us that we would not see a significant performance difference due to the requirement that the OO1 benchmark be run remotely. A Lisp version would have had better performance, according to Itasca support, only if it was allowed to run exclusively in the native (local) Lisp API.

Itasca support examined our benchmark code on several occasions and helped to correct several problems with our program. The next section discusses the problems we encountered implementing the OO1 benchmark in the Itasca DBMS.

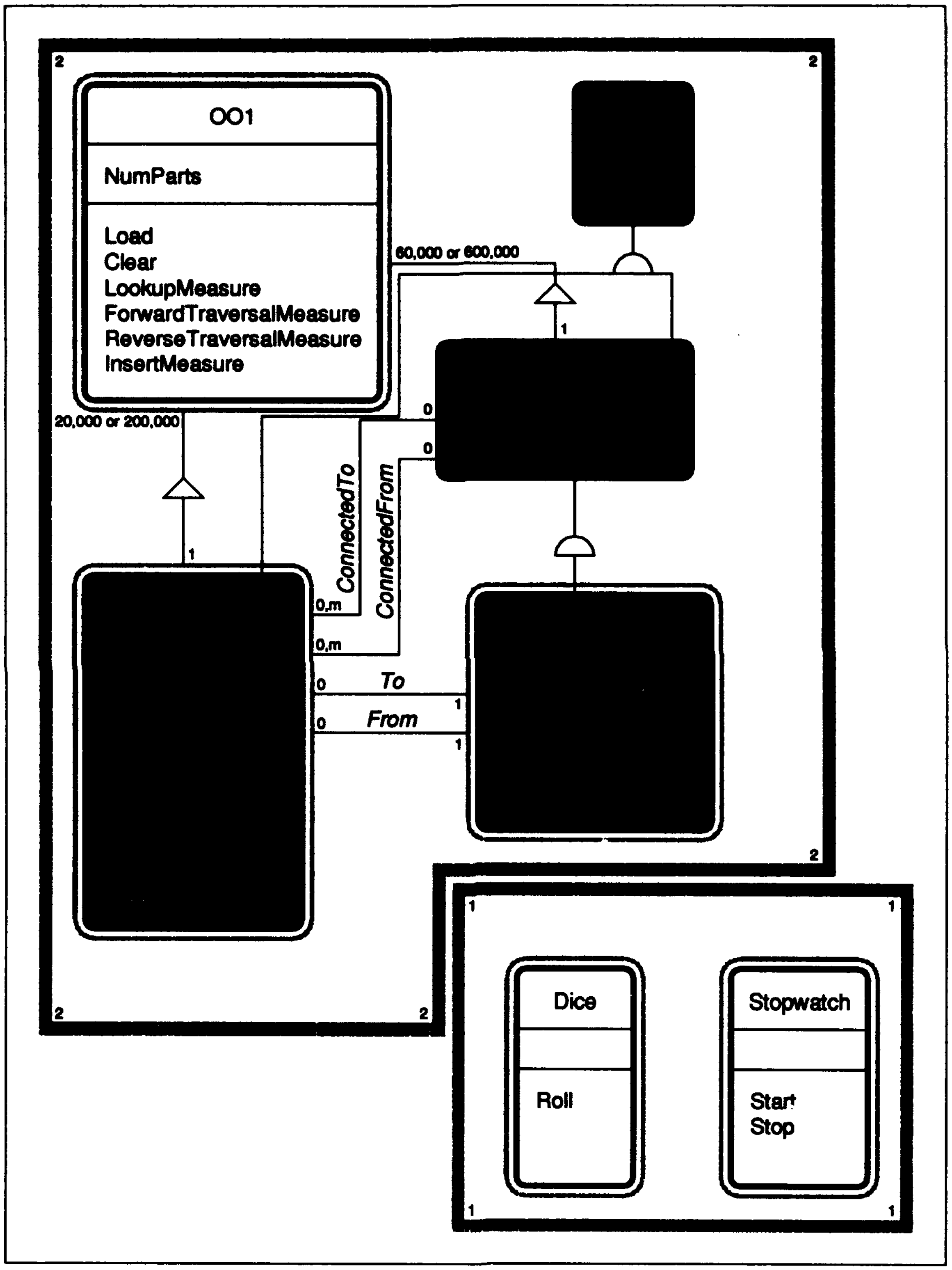

Figure 15. Itasca OOD Diagram for the OO1 Benchmark

The Itasca version of the OO1 benchmark took the most time to complete. A large number of problems were encountered. The problems encountered with the Itasca implementation fall into two groups: problems encountered with the Itasca DBMS and problems with the C++ API. The reason we note the difference is that the C++ API is a new product for Itasca and will no doubt change a good deal in the near future, while the Itasca DBMS should be more mature. Most of the difficult and time consuming problems encountered with the Itasca implementation involved trouble with the C++ API and not the Itasca DBMS. The following is a list of the problems we encountered with the Itasca DBMS and C++ API:

- *Slow Database Commit*: The commit of data takes a significant amount of time in the Itasca database. Itasca support indicated that they have been working on the time the database commit takes, which they report is mostly taken up by object hashing and writing to the disk. They report that they were able to improve the performance by 400%, but at a cost of significant increases in memory use. We hope that future versions of Itasca will solve this performance flaw.
- *Unbounded Growth of the Database Log File*: Due to the limited amount of disk space available to perform our research we encountered a problem with the Itasca database log file. The log file stores the transactions which are in progress. In theory, the log file should shrink when database commits are performed, but the current version of Itasca only garbage collects the log file during a restart of the database. To shrink the size of the log file, the database must be shutdown and restarted. The expanding log file filled the entire disk on our test workstation on several occasions.
- *Flat Name-Space for Database Objects*: The name-space in the Itasca database used for class names is flat across all users of the database. Only one class in the database may have the same name. This can cause conflicts with existing names. For the OO1 benchmark, all the persistent class names were prefixed with the string *Oo1* to avoid this problem.
- *Private Database Numbering*: All private databases are assigned a number when they are created. The user has no control over private database number assignment

but is responsible for keeping track of these numbers. In Itasca each user contains an attribute which defines the private databases which have been created by that user. When a user connects to Itasca (in a program or using one of the Itasca GUI tools) Itasca places that user in the first private database that exists at the current site and in the list of private databases created by that user. If there are no private databases which have been created by the user, then the user is placed in a private database to which the user has been granted access. If a user is not allowed in any private database, then the user is placed in private database -1 (which means no private database). The private database system was difficult to decipher when we first started work with Itasca and often placed our benchmark program in the wrong private database. There is currently no logical or symbolic representation for the private database numbers and a user is responsible for keeping track of all the private database numbers they own.

- *No Circular Class References in the C++ API*: The Itasca C++ API could not represent circular references between classes. For example, if class A has a reference to class B then class B is not allowed to have a reference to class A. This limitation is only in the Itasca C++ API and is an unusual limitation. The C++ language does not have this limitation and the Itasca database does not have the limitation. Itasca customer support told us that the problem would be cleared up in the next release. To solve the problem, a dummy-superclass was declared for the *Oo1Connection* class named *Oo1AbstractConnection*. The *Oo1Part* class contained references to the *Oo1AbstractConnection* class rather than the *Oo1Connection* class and the *Oo1Connection* class contained the other side of our reference with a reference to the *Oo1Part* class. To the application program the end result appears as if there is a circular reference, but there is some overhead incurred because of the use of the superclass.

- *Lack of Non-Persistent Method Support in the C++ API*: The Itasca C++ API does not allow non-persistent methods or normal C++ class methods to be easily created. The Itasca *dynamic schema editor* (DSE) is the preferred tool to create a database schema for Itasca. Once a schema is created in the Itasca database, the DSE is used to automatically generate a C++ header file. Since the DSE does not allow the creation of C++ methods inside of classes, the generated C++ header file must be edited to add the methods. Itasca support recommended using a C include file (a ".h" file) to define the methods and including the file in the generated C++ header file. This was done, and as Itasca support indicated, it saved a good deal of effort when the C++ header file was regenerated.

A related problem to the inability to allow non-persistent methods was the lack of the ability to declare attributes in a class private to the class. This is an ability of the C++ programming language which was not included in the Itasca C++ API. Itasca support indicated that this was not supported because there is no concept of the private class member in the Itasca database.

- *Transient Memory Management*: A large problem encountered with the Itasca C++ API was a misunderstanding on our part as to how the API handled transient memory associated with persistent objects. When an object is read from the database, some transient memory is allocated to hold that object. In Itasca, the management of that memory is not considered the responsibility of the DBMS but of the application. Though this is not documented in the C++ API manuals [20], the following rules (which we deduced) were confirmed with Itasca support:

  - If a persistent object is created via the C++ *new* operator, then the application is responsible for its transient memory.
  - If a persistent object is retrieved via an Itasca query, then the application is responsible for its transient memory.
  - If a persistent object is retrieved via an Itasca iterator, then the application is not responsible for its transient memory.

  This was a very confusing portion of the Itasca C++ API and caused us to create huge memory leaks in our benchmark program. The above rules were only deduced after Itasca support removed the majority of the memory leaks from our program. Itasca support informed us that transient memory management should have been in the manual and that the documentation oversight will be corrected in the future.

4.3.2 Matisse Implementation. Our OO1 implementation for the Matisse DBMS was written in the C++ programming language, but the object-oriented services API to Matisse only uses the C programming language subset of C++. Three versions of the benchmark were written for Matisse. We first examine the benchmark implementation in Matisse and then cover the problems encountered during implementation.

Two executable programs were created for the Matisse version of the OO1 benchmark. The first program was the *bench* program and the second was the *oo1schema*. The bench program implemented the benchmark operations and the oo1schema program loaded or removed the database schema from the Matisse DBMS.

The Matisse database makes no attempt to be transparent to the application program. The bindings to the object oriented services are a strictly functional API written in the C programming language. The object-oriented nature of the Matisse database is inside the database and is completely different from the object-oriented nature of the C++ programming language. Therefore, persistent objects in Matisse are not implemented as C++ classes. The persistent objects are represented only in the database and communication takes place through the Matisse object-oriented services API. The non-persistent methods which are needed for the benchmark were implemented as C functions with calls to the database.

Matisse support examined our implementation on several occasions and made several suggestions for improvement. The various improvements made based upon their input will be covered in the next section.

Most of the problems encountered during the development of the Matisse implementation involved difficulty in understanding the operation and use of the Matisse DBMS. The following is a list of the problems we encountered with the Matisse DBMS:

- *Version Collection*: The Matisse system performs intrinsic versioning of all objects. If an object is changed in the database, a new version is created. This scheme requires a large amount of disk space. To collect old versions and to compact the current objects in the database, a program called *mt_collect_versions* must be used. This program starts up a task in the database which runs concurrently with all other database tasks. This program needs to be run rather frequently or a large amount of disk space is unnecessarily consumed. Due to the limited amount of disk space available for our test workstation we frequently ran out of database disk space (or *silo* space in Matisse terms).

The collect versions operation occurs in asynchronous mode and the Matisse server creates a dedicated thread to perform it. Thus, there was no easy method to measure the time it takes to execute. The collect versions operation has two main goals: collecting the old versions of the objects and compacting objects into the minimum number of buckets. Matisse support informed us that most Matisse customers create a *crontab* entry to run the collect versions in a periodic fashion.

Because the version collection program runs concurrently with other database tasks and can not be measured, we were unable to determine the exact impact which this has on database performance. We ran several tests with different delays between using the version collection and running the benchmark. It appears to be very important to analyze the application for which the database is being used and set up a reasonable schedule for the use of the version collection program. If any application is running at the same time the version collection program is running, the other application has priority. Therefore, database "down" time must be planned to run the version collection program.

- *Schema Creation*: The first version of our OO1 benchmark implementation for Matisse created the database schema during the load operation and removed the schema during the clear operation. Customer support pointed out to us that this was causing some additional overhead of which we were unaware. The Matisse object services provides two different libraries which applications can link to: the *data management* (DS) library and the *data and schema management* (DE) library. The DS library is used for applications which modify the schema in the Matisse database. The DE library is used for applications which work with objects in the database but do not change the schema. There is some additional transaction overhead when the DS library is used to ensure that schema changes are done properly. Matisse support had run our original program to load the data, then relinked with the DE library to run the benchmark measures and suggested that we do the same. Because the OO1 benchmark does not require schema evolution, we decided to separate the schema creation from the benchmark program entirely. A new program, called *oo1schema*, was created to load the benchmark schema into the Matisse database. This program was linked with the DS library and the benchmark program, *bench*, was linked with the DS library. The new program made the Matisse implementation closer to our Itasca and ObjectStore implementations. For the Itasca implementation, the schema is created using the *dynamic schema editor*, not by the benchmark program. For the ObjectStore implementation, the schema is created by the OSCC compiler while the program is being compiled and linked.

- *Use of Client Memory Transport for Local Clients*: The Matisse DBMS allows the use of memory transport for local clients to the database, rather than the use of network transport. Memory transport is designed to improve local client performance by using shared memory and semaphores. Matisse support described to us the method for enabling local client memory transport and we encountered no problems with it. However, we feel that this setting should be the default for local clients, not an esoteric parameter in the Matisse configuration file.

- *Non-Blocking Read-Only Transactions*: A feature of Matisse derived from the intrinsic versioning is the ability to perform a non-blocking read-only transaction on any old version of an object. Matisse support recommended using this feature for the lookup and traversal measures of the OO1 benchmark since they are read only. We did not use this ability in our OO1 benchmark implementation because all the other implementations used full transactions, but do note that it is a powerful feature of the Matisse DBMS.

- *Object Identifier (OID) Usage*: Matisse support informed us that OIDs in Matisse are valid over the entire lifetime of an object. Therefore, it is efficient to read in all the OIDs necessary for a program and then use them where necessary. We had been looking up OIDs during each transaction.

- *Benchmark Design*: Matisse support was the only vendor support group which questioned our design of the OO1 benchmark. The basic problem which they had was with our use of two inverse relationships between the part and connection classes. They felt that only one inverse relationship was necessary. They also noted that the specific cardinality in our object model was not used in the implementations. In the OO1 benchmark there are always three connections from a part, but the implementations allowed this number to be larger or smaller. These questions from Matisse support were resolved through the communication of our design to them.

4.3.3 ObjectStore Implementation. Our OO1 implementation for the ObjectStore DBMS was written in the C++ programming language using the ObjectStore DML extensions to the language. Eight versions of the benchmark were written for ObjectStore. No changes to the final design were necessary for the ObjectStore version of the OO1 benchmark.

Development of our benchmark with the ObjectStore DBMS was the closest to simply developing the benchmark in the C++ programming language. ObjectStore appears to the developer as a persistent C++ implementation with additional support for database constructs. Persistent classes in ObjectStore are no different from transient objects in C++, except for the method of their creation.

ObjectStore support was offered the opportunity to examine the benchmark program we developed but they declined. They did provide a good deal of help with specific problems encountered during development but did not provide much input on improving the performance of the final program.

The following is a list of the problems we encountered with the ObjectStore DBMS:

- *Persistent Object Leak*: The greatest problem with the ObjectStore version of the OO1 benchmark was a persistent object leak. This problem had occurred with all the implementations, but it was discovered first in the ObjectStore implementation of the benchmark. In a programming language, it is very important for a program to give back dynamic memory to the operating system. If this is not done then the program is said to have a *memory leak*. With an object-oriented DBMS it is possible to have a memory leak into the persistent storage area. We have called this a *persistent object leak* (since objects are lost in the database not memory).

In our OO1 implementation, the persistent object leak occurred during the insert measurement. The insert was creating 100 new part objects and 300 new connections objects, but was only deleting the 100 part objects. Thus, every benchmark run on the database was losing 300 connection objects in the depths of the database. The problem was detected when the reverse traversal measurement started crashing the benchmark program. The lost connections maintained a connection to the part to which they were connected, so that lost connections could be traversed during the reverse traversal measure. When a lost connection was traversed, the application found that the connecting part did not exist and would crash. The C++ destructor functions of the part and connection classes were modified to ensure that all the inserted objects were deleted.

Persistent object leak could become a major problem with object-oriented DBMSs, and we noted that none of the vendors have provided tools to assist the application developer to find lost objects.

- *DBMS Tuning Parameters*: We found the number of performance tuning parameters available in the ObjectStore DBMS staggering. Despite this large number of parameters, ObjectStore does not provide a guide to database performance. A large amount of time can be spent tuning an ObjectStore database application for performance, and this accounted for the the eight different versions of the benchmark we created. We saved a great deal of time by examining an ObjectStore OO7 benchmark implementation developed at the University of Wisconsin-Madison and used many of the same performance settings [7].

- *Index Use During Queries*: Unlike current relational databases, an index must exist for it to be used for a query. The query optimizer in ObjectStore will not create an index if you have not defined one. This also seemed to be the case in Itasca. Matisse provides no support for a query language which can be used in an application.

4.4 Verification of Benchmark Program Correctness

It was very important that the implementations of the OO1 benchmark be correct. Two methods were used to verify the benchmark implementations: *code inspections* and *testing*.

Code inspections were used for two purposes: *error identification* and *implementation consistency*. First, the inspections identified errors in the benchmark implementations. Second, the inspections identified variations in the source code of the three implementations. Every attempt was made to keep the implementations consistent and the inspections avoided unnecessary deviations in the implementations. A code style guide developed for this research helped to simplify inspections of the source code. The style guide also provided a consistent style for all the implementations. The code style guide is documented in Appendix G.

A large amount of testing was done on the benchmark programs. Testing was done on very small databases (10 parts with 30 connections) and used debug code developed into all the source code of every benchmark implementation. A consistent style of debug output was obtained through the use of a macro package developed by Microsoft [27]. This package is documented in Appendix G.

Additional verification was provided by the DBMS vendor support groups, who examined and executed the benchmark implementations at their sites. An exception to this was ObjectStore, who never examined the source code of our benchmark implementation for their database.

4.5 Summary

This chapter has covered the analysis, design, and implementation of the OO1 benchmark developed for this research. We have examined, in detail, the problems encountered when working with each of the three commercial object-oriented DBMS. Chapter VI examines the results obtained from our runs of the OO1 benchmark. The next chapter, Chapter V, examines the simulation benchmark developed for this research. The simulation benchmark investigates the ability of the three commercial object-oriented DBMSs to support simulation systems.

V. *Simulation Benchmark*

In this chapter we examine the requirements, design, and implementation of the simulation benchmark. The simulation benchmark is a new benchmark designed for the computer simulation domain. The simulation benchmark sets up a complete simulation system in an object-oriented DBMS. The purpose of the benchmark is to examine the performance and functional capabilities of the three commercial object-oriented DBMSs available at AFIT for use with computer simulation.

5.1 Benchmark Description

This section describes our simulation benchmark for object-oriented DBMSs. The benchmark simulates aircraft searching for moving trucks over an area of land. When an aircraft finds a truck, the location and the time when the truck was found is logged. The simulation used in the benchmark is a *stochastic discrete-event simulation model* [24]. The benchmark requires an entire simulation system to be built for each object-oriented DBMS which is tested. The object-oriented DBMS must be used in all portions of the simulation system for storage, including simulation execution. This benchmark attempts to re-draw the traditional line between the simulation software and the database (traditionally implemented as flat files). We examine the requirements of the benchmark simulation system. Then the benchmark measures are described; and, finally, we provide some justifications for our benchmark.

5.1.1 Benchmark Requirements. The software required for our benchmark consists of several parts: a model (of the aircraft and the trucks, and a map), a means to configure a scenario, a simulation executive (to control simulation execution), a post processing module to view simulation results, and support libraries (which provide support for both simulation and benchmarking). The benchmark quantitatively measures performance in all the areas of the simulation system. The benchmark is also a *functional* benchmark since we examine the qualitative capability of an object-oriented DBMS to support a general simulation system.

Before examining specific requirements of the benchmark software, we enumerate several general rules to which the benchmark must adhere. These rules apply to every portion of the benchmark software.

1. *Persistence of the Simulation Model*: The simulation model used in the simulation benchmark must be persistent. The model must be saved between one run of the benchmark program and another.
2. *No Flat-Files*: The benchmark software may not use flat-files for data storage at any point in time; the object-oriented DBMS must be used. This ensures we are measuring the objected-oriented DBMS's ability to store data, not the operating system's.
3. *The Object-Oriented DBMS Must Manage the Cache*: The benchmark software must allow the object-oriented DBMS to decide when data is brought in from the disk. The benchmark program may not read all data from the object-oriented DBMS, run the simulation, and then save all data to the DBMS. This ensures we are measuring the objected-oriented DBMS's ability to cache data, not the benchmark programmer's.
4. *Multi-User Capability*: The benchmark implementation must be able to provide multi-user support. For example, two copies of the simulation system must be able to execute at the same time inside the same database. Several of the benchmark measures depend on this ability. The object-oriented DBMS's support for transactions, concurrency, versioning, and locking should be used to provide the multi-user capability, not custom code written by the benchmark programmer.
5. *Use Common Graphics Code*: All benchmark implementations must use the same graphics code for the user interface. The goal of the benchmark is to measure the performance of the object-oriented DBMS, not to examine the performance of a graphics library. But it is important, from a functional point of view, to note the ability of the object-oriented DBMS to work with the graphics library.

Our intent is that a single program be built to represent the simulation system. The program must provide a windowing-system user interface to the simulation system. To support the benchmark measures, the program must be instrumented with routines to measure elapsed time and throughput.

5.1.1.1 Simulation System and Model Capabilities. This section describes the requirements of the simulation environment for the simulation benchmark. All of the requirements described in this section must be supported by the benchmark program. The benchmark requirements are as follows:

- *Connect and Disconnect to an Object-Oriented DBMS*: The benchmark program must provide a method for the user to connect to an object-oriented DBMS. The program should prompt the user for any authentication information required by the DBMS to grant access.
- *Benchmark Simulation Model*: The benchmark supports a single model, that of aircraft searching for trucks on a map. Since the benchmark is a discrete-event simulation, the simulation state evolves over time with changes occurring instantaneously at selected points in time [24]. The changes in simulation state are triggered by *events*. The simulation contains two type of simulation objects: *active* and *passive*. Active simulation objects execute events, while passive objects do not. The aircraft and trucks are active simulation objects, and the map, which is a board of hexes, is passive. We will first describe the state information, or attributes, required by each object in our model, then describe the events required by the model.

  The aircraft in the simulation each maintain the following information:

  - *Aircraft Name*: This value is a string of 9 characters which uniquely identifies an aircraft. The string is in the format "`AIR-`*nnnnn*". The *nnnnn* represents a unique number for the aircraft. Numbers are assigned sequentially to each aircraft starting from a value of 0.
  - *Location*: All aircraft know where they are located. The value is a hex identifier specifying a single hex on the hex board.
  - *Home Base*: This value is a string of 10 characters. The value is randomly selected from the strings {"`HOME-BASE1`"... "`HOME-BASE9`"}.

- *Miscellaneous Simulation Data*: This value is a string of 50 bytes. The data must be selected from at least ten different values and represents additional information which would be required by a more complex simulation system than is represented in this benchmark.

The trucks in the simulation each maintain the following information:

- *Truck Name*: This value is a string of 9 characters which uniquely identifies a truck. The string is in the format "`GND-`*nnnnn*". The *nnnnn* represents a unique number for the truck. Numbers are assigned sequentially to each truck starting from a value of 0.
- *Location*: All trucks know where they are located. This value is a hex identifier specifying a single hex on the hex board.
- *Type*: This value is a string of 11 characters. The value is randomly selected from the strings {"`TRUCK-TYPE1`" ... "`TRUCK-TYPE9`"}.
- *Payload*: This value is a string of 11 characters. The value is randomly selected from the strings {"`CARGO-TYPE1`" ... "`CARGO-TYPE9`"}.
- *Miscellaneous Simulation Data*: This value is a string of 50 bytes. The data must be selected from at least ten different values and represents additional information which would be required by a more complex simulation system than is represented in this benchmark.

There are three different types of events required by the simulation benchmark model. They are the following:

- *Aircraft Move*: This event moves an aircraft into a randomly selected adjacent hex.
- *Search*: This event causes an aircraft to search the hex in which it is located for any trucks. If any trucks are found they are logged, so that they may be reported in the summary report.
- *Truck Move*: This event moves a truck into a randomly selected adjacent hex.

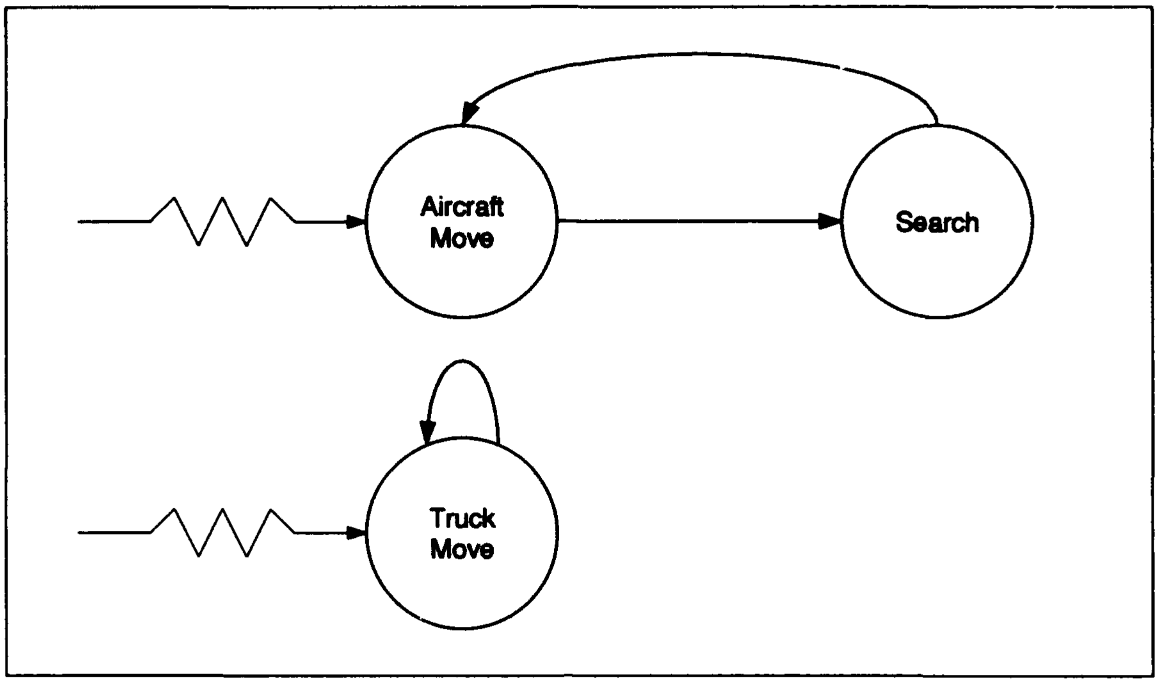

Figure 16. Event Graph (Queuing Model) for the Simulation Benchmark

The *Aircraft Move* and *Search* events are sent to aircraft and the *Truck Move* is sent to trucks. An *event-graph* of the events is shown in Figure 16. The event-graph notation is defined by Law in [24]. Each node of the graph represents an event type. Each arc represents how an event may be scheduled by another event or by itself. In the simulation benchmark, the *Aircraft Move* event schedules a *Search* event, the *Search* event schedules an *Aircraft Movement* event, and the *Truck Move* event schedules another *Truck Move* event. The graph indicates that the *Aircraft Move* and *Truck Move* events must be scheduled initially for each aircraft and truck. All events are scheduled in the future based upon a random draw from an exponential distribution. For the *Aircraft Move* event, a mean value of 60 seconds is used; for the *Search* event, a mean value of 30 seconds is used; and for the *Truck Move* event, a mean value of 600 seconds is used.

- *Create, Store, and Copy Simulation Models*: The benchmark program must be able to create and manage several simulation models. For simplicity, only the search model defined above is required, but multiple distinct instances of this model must be allowed in the database (each instance having a separate simulation state). Each model instance is created by the user and assigned a name. The user is also allowed to make a copy of any existing model.

- *Configure a Scenario*: The benchmark program must be able to configure a scenario within a simulation model. To create a scenario, the user enters the number of aircraft, the number of trucks, and map size desired for the scenario. The user can also remove all the scenario elements from a model, effectly destroying the scenario.

- *Support a Simulation Time Slice*: The user is able to define a simulation time slice. The time slice defines how many seconds of time are simulated in a single step of the simulation, although many simulation events (or none) may be simulated during a single time slice. The time slice defines the granularity of the simulator. The time slice is defined as a time in seconds and must have a value of 1 or greater. The time slice defines the *fixed-increment time advance* for the simulation benchmark program [24].

  The time slice is used to define the benchmark's transaction model. Each time slice is executed as a single transaction in the database. This definition of the transaction model allows us to vary the amount of work done during a single transaction.

- *Support Batch and Real-Time Simulation Execution*: The benchmark program is able to execute the program to a goal time, running as fast as the system will allow, although the simulation must still observe the time slice setting. This type of execution is called batch execution. The program must also support real-time execution at a user specified ratio with wall clock time. The user is allowed to specify a *time ratio* for the simulation to execute at. The time ratio is defined as the ratio of wall clock time to simulation time. To simulate at the specified time ratio, the simulation executes a single time slice, then delays for the remainder of the duration specified by the time ratio. For example, if the time slice is set to 60 seconds, and the time ratio is set to 1, then if the simulation requires 4 seconds to simulate the 60 second (simulation time) time slice, the simulation would delay for 56 seconds before the next time slice was executed. If the time ratio were set to 2 the delay would be 116 seconds. And if it were set to 0.5, then the delay would only be 26 seconds.

- *Support Post-Processing of Simulation Data*: The benchmark program provides two types of post-processing. First, a graphical hex map displaying all the aircraft and trucks in the current model is available at user demand (the map image is not required to be stored in the DBMS). Second, a summary report of all the trucks found by aircraft since the simulation started may be generated on demand. The summary report is generated to a text file. This summary report contains the following information for each truck found:

  - *Report Time*: The simulation time when the report was generated.
  - *Aircraft Name*: The name of the aircraft which found the truck.
  - *Truck Id*: The name of the truck found.
  - *Location Found*: The location of the truck (the hex identifier) when it was found.
  - *Time Found*: The simulation time when the truck was located.

This section has examined the requirements of the simulation benchmark. The next section describes the operations which our benchmark measures.

5.1.2 Benchmark Measures. This section examines the quantitative measurements required by the simulation benchmark. These measurements are done for two different database sizes. The first database contains 1,000 trucks, 500 aircraft, and a 50×50 hex board. This database is called the *small* database. The second database contains 10,000 trucks, 5,000 aircraft, and a 100×100 hex board. This database is called the *large* database. The following benchmark measures are made on the simulation system:

- *Model Creation*: Measure the elapsed time required to create a new model instance. This measurement does not include the time to create the scenario. This is a measure of elapsed wall clock time.
- *Scenario Creation*: Measure the elapsed wall clock time required to create the scenario.
- *Simulation Execution (Hour Run)*: Use the *run-until* function of the simulation executive to run the simulation for 1 hour with the simulation time slice set to 60

seconds. This is a measure of elapsed wall clock time. This measure is also run with a time slice of 600, 1800, and 3600.

- *Simulation Throughput*: Run the simulation at the fastest ratio possible with the time slice set to 60 seconds. The ratio must be at steady state for at least 10 minutes of real time. Report the time ratio obtained. 90% of the time slices must maintain the reported ratio.
- *Version Creation*: Measure the elapsed time to create a new version of a model which has been run through time for 1 hour. The new version may be a complete copy of the model or a new version created by the object-oriented DBMS.
- *Map Creation*: Measure the elapsed time for the creation of the map in the post-processing portion of the simulation system. This measurement is taken for a paused simulation, a running simulation, and a simulation running on a remote computer.
- *Report Creation*: Measure the elapsed time for the creation of the summary report containing a list of the trucks found by aircraft during simulation execution.

This section has described the quantitative benchmark measures. In addition to the quantitative measures, the functional ability of the DBMS in which the benchmark is implemented to support the simulation should be noted. The next section describes the justification for our benchmark.

5.1.3 Benchmark Justification. Several choices were made in developing a meaningful benchmark for the simulation domain. This section examines our reasons for the current benchmark and attempts to justify them as much as possible. To the best of our knowledge, this is one of the first attempts to build a persistent simulation environment using an object-oriented DBMS; the only other work in this area we have encountered is described in [40].

We feel that the model implemented in this benchmark provides several of the elements found in most discrete-event simulation systems. While it is true that the actions of the simulation objects, the aircraft and trucks, are not very interesting, they do provide a constant load on the simulation system when it is running. A problem with the use of objects which move so regularly is that not all simulations exhibit this property. For example, it is possible that a simulation may be very inactive for a long period of time, then be required to process a large number of events in a short period of time. While this is possible, we felt it was much more important to provide a predictable load (on average) so that it was possible to understand how many events the simulation would execute during a period of time. With this understanding it could be possible to use our results to predict performance of another simulation based upon the expected number of events required to be executed during the simulation.

It is also important to note that the queue of events pending for the simulator is going to be of the same magnitude as the number of active objects (aircraft and trucks) in the simulation. This is due to the way events are scheduled (see Figure 16). Here we also note that not all simulation systems exhibit this property.

The inclusion of a full graphical user interface (GUI) as a required part of the benchmark could be controversial, but we feel that any modern simulation system is going to require a graphical interface which is well integrated with the simulator. Therefore, it is essential to measure the object-oriented DBMS's ability to interface with the GUI while supporting the simulation system.

An earlier version of the benchmark included terrain data for each hex on the hex board, but we decided to eliminate this. The terrain data complicated implementation and did not provide enough benefit to justify keeping it. The same amount of data can be created in the database by creating more aircraft and trucks in a scenario.

We have provided the map and the summary report to represent two types of information which a simulation system would have to keep: current and cumulative. The map reflects the current state of the simulation and requires that all the objects in the simulation be traversed. The summary report requires that information which was available in the past be maintained, via a logging method of some sort, and reported. Both types of information are required in typical simulation systems.

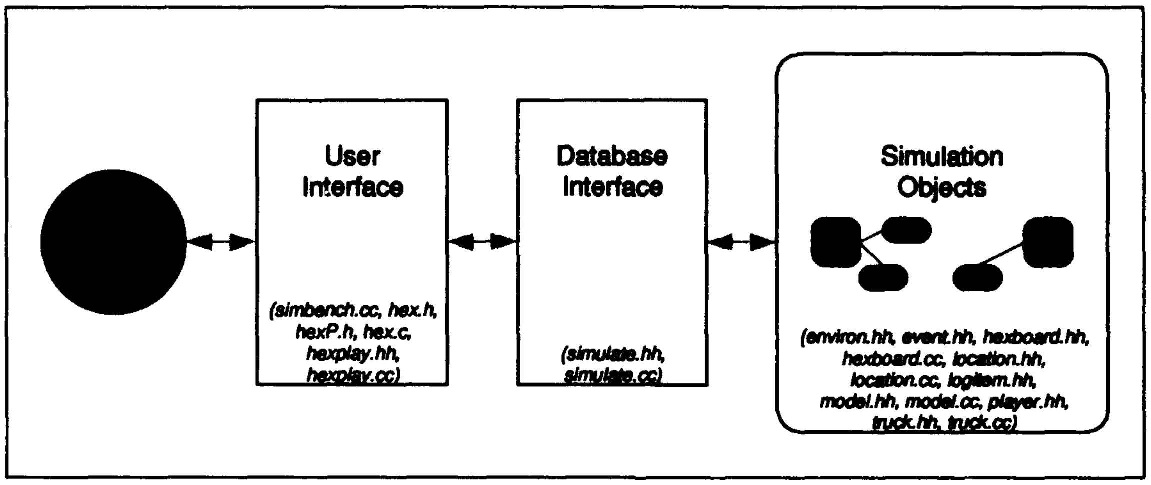
User Interface
(simbench.cc, hex.h, hexP.h, hex.c, hexplay.hh, hexplay.cc)
Database Interface
(simulate.hh, simulate.cc)
Simulation Objects
(environ.hh, event.hh, hexboard.hh, hexboard.cc, location.hh, location.cc, logitem.hh, model.hh, model.cc, player.hh, truck.hh, truck.cc)

Figure 17. Simulation Benchmark Design—Big Picture

5.2 Simulation Benchmark Design

For our design of the simulation benchmark, as in the design of the OO1 benchmark, we used the methods described by Coad and Yourdon [11].

The big picture of our benchmark design is shown in Figure 17. The user of the benchmark program interacts with a graphical user interface. The user interface interacts with a database interface module which hides the specifics of the database from the user interface implementation. The database interface interacts with the simulation objects based upon commands from the user interface. All the simulation objects are required to be persistent and are therefore stored in the object-oriented DBMS. The next sections describe our design of the human interaction component (the user interface), the problem domain component, and finally the database interface.

5.2.1 Design of the Human Interaction Component. Unlike the OO1 benchmark, the simulation benchmark requires a large user interface component. Since we developed our implementations on UNIX workstations, we decided to develop the benchmark user interface with the OSF/Motif graphical user interface (GUI). Our design and implementation of the user interface were designed to be compliant with the *OSF/Motif Style Guide* [29], with additional ideas from Heller [17]. In this section we overview our user interface design.

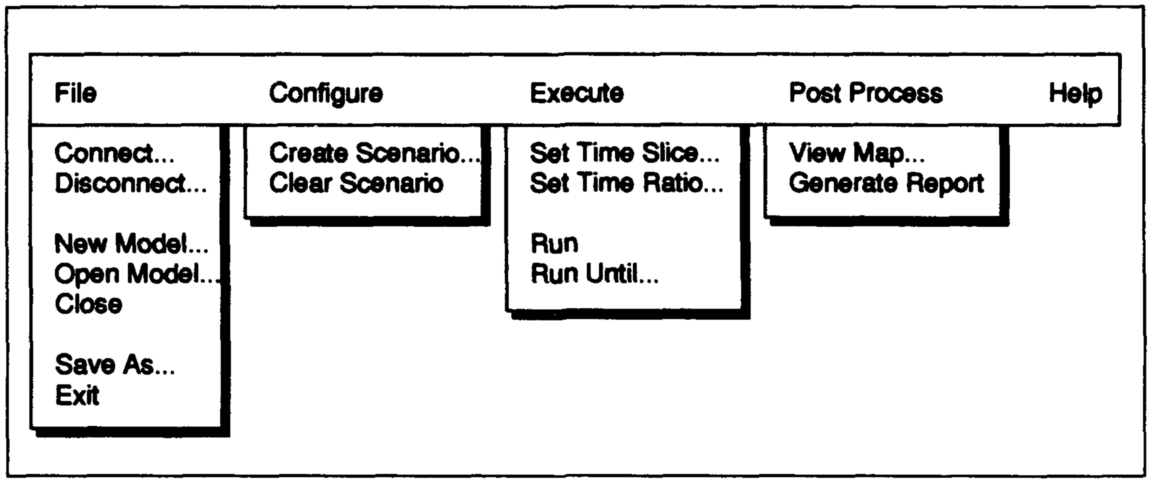

Figure 18. Simulation Benchmark User Interface Menu

The main menu for our user interface to the simulation benchmark program is shown in Figure 18. The main menu is the primary interface for the program user. The user is allowed to select any menu item which is not dimmed (menu choices are only allowed when they have meaning to the program). The following menus appear on the menu bar:

- *File*: Contains commands to interact with the object-oriented DBMS, to create, copy, select, and close simulation models, and to exit the program. The choices on this menu are described below:
  - *Connect*: This choice allows a user to connect to the object-oriented DBMS. The user is prompted for any authentication information required by the DBMS. Then a connection is established.
  - *Disconnect*: This choice terminates the program's connection to the object-oriented DBMS.
  - *New Model*: This choice creates a new model instance in the database. The user is prompted for a name for the model. Then the model is created. The name for the new model must not already be the name of an existing model. After the model is created, it is set as the current model.

- *Open Model*: This choice presents the user with a list of the existing models in the database. The user selects one of the choices and that model is set as the current model.
- *Close*: This choice closes the current model.
- *Save As*: This choice creates a copy of the current model. The user is prompted for a name for the copy, and then a copy is made. The current model remains the same. If the user wishes to work with the new copy, the current model must be closed. Then the copy must be opened with the File | Open Model command.
- *Exit*: This choice exits the program.

• *Configure*: Contains commands to build and clear a scenario for the current model. The choices on this menu are described below:

  - *Create Scenario*: This choice creates a scenario for the current model. The user is prompted for the number of aircraft, the number of trucks, and the size of the hex board desired. Then the a scenario with the correct number of trucks and aircraft is created. If a scenario already exists in the current model, it is cleared before the new scenario is created. The simulation time for the current model is set to time 0.
  - *Clear Scenario*: This choice clears the current model's scenario.

• *Execute*: Contains commands to control the execution of the simulation. The choices on this menu are described below:

  - *Set Time Slice*: This choice sets the value of the time slice for the simulation. If the time slice is not set by the user, it defaults to a value of 1.
  - *Set Time Ratio*: This choice sets the value of the time ratio for the simulation. The time ratio is the ratio of wall clock time to simulation time. If the time ratio is not set by the user, it defaults to a value of 1.
  - *Run*: This choice starts running the simulation at the set time ratio. A dialog appears to the user reporting the current simulation time and statistics about the execution of the simulation. The dialog is updated at the end of each

```
-------------------------------------------------
AFIT SimBench REPORT (SIM TIME: 300 sec)
-----------+------------+------------+-----------
 AIRCRAFT |       TRUCK |  LOCATION |   SIM TIME
-----------+------------+------------+-----------
AIR-00008 |  GND-00022 |  0016-0020 |     51 sec
AIR-00039 |  GND-00043 |  0041-0039 |     79 sec
AIR-00020 |  GND-00004 |  0013-0032 |    119 sec
AIR-00037 |  GND-00018 |  0005-0017 |    160 sec
-----------+------------+------------+-----------
```

Figure 19. Example of the Summary Report

executed time slice. The dialog contains a push button which allows the user to stop the simulation at any time.

- *Run Until*: This choice runs the simulation, as fast as possible, to a specific goal time. The user is prompted for the goal time, which must be in the future, and the simulation is executed until that goal time. This option also displays a dialog which is updated after each time slice. The user may terminate the run until operation at any time by pressing a stop button which is displayed in the dialog.

• *Post Process*: Contains commands to execute the two post-processing options of the benchmark. The choices on this menu are described below:

  - *View Map*: This choice displays a hex board to the user with the aircraft and trucks in the current model plotted on the map. The map may be left up and updated by a user using an update push button or it may be dismissed at any time. An update of the map is capable of responding to changes in the scenario, including a change in the hex board size.

  - *Generate Report*: This choice generates a summary report to the text file *SIM_REPORT*. If the file already exists, it is overwritten. A sample of the summary report is seen in Figure 19.

- *Help*: Contains help about the simulation benchmark. The simulation benchmark does not provide a full help system due the large amount of effort required to implement such a system in the current version of OSF/Motif.

5.2.2 Design of the Problem Domain Component. This section describes our design for the problem domain component of the simulation benchmark. The problem domain component consists of all the simulation objects required by the simulation benchmark. An OOD diagram of our design for the problem domain component of the simulation benchmark is shown in Figure 20.

The *model* class is a container class which holds all the components of a model. Every instance of the model class has a name which is stored in the *name* attribute. A model contains a simulation *environment*, *events*, and *logitems*. The simulation environment contains the active and passive simulation objects. The model's events define the future actions of the active objects in the simulation environment. The logitems record information which occurred during the simulation for later reporting. An instance of the model class, with all its parts, defines a simulation state. The model stores its current simulation time in the *SimTime* attribute and is able to advance simulation time by dispatching events to the active objects in the environment. Simulation time in a model is advanced using the *RunUntilSimGoalTime* service. This service executes a sequence of events in time order until the goal time is reached. The concepts of time slice and time ratio are not understood by the model class; it is only able to advance to a specific future time. The *Schedule* service of the model class allows active object in the model's environment to place schedule future events for the model. The *AddEnvironmentMember* service allows new objects to be added to the model's environment.

All objects which are part of a model's environment are subclassed from the *environment* class. The environment class is a virtual class. That is, no instances of the class are allowed to be created. The main purpose of the environment class is to force every subclass to implement two services: a *clone* service and a *query* service. The clone service creates an exact copy of the object. This service is required to allow distinct copies of models to be created. The query service provides a means for other objects to obtain information about

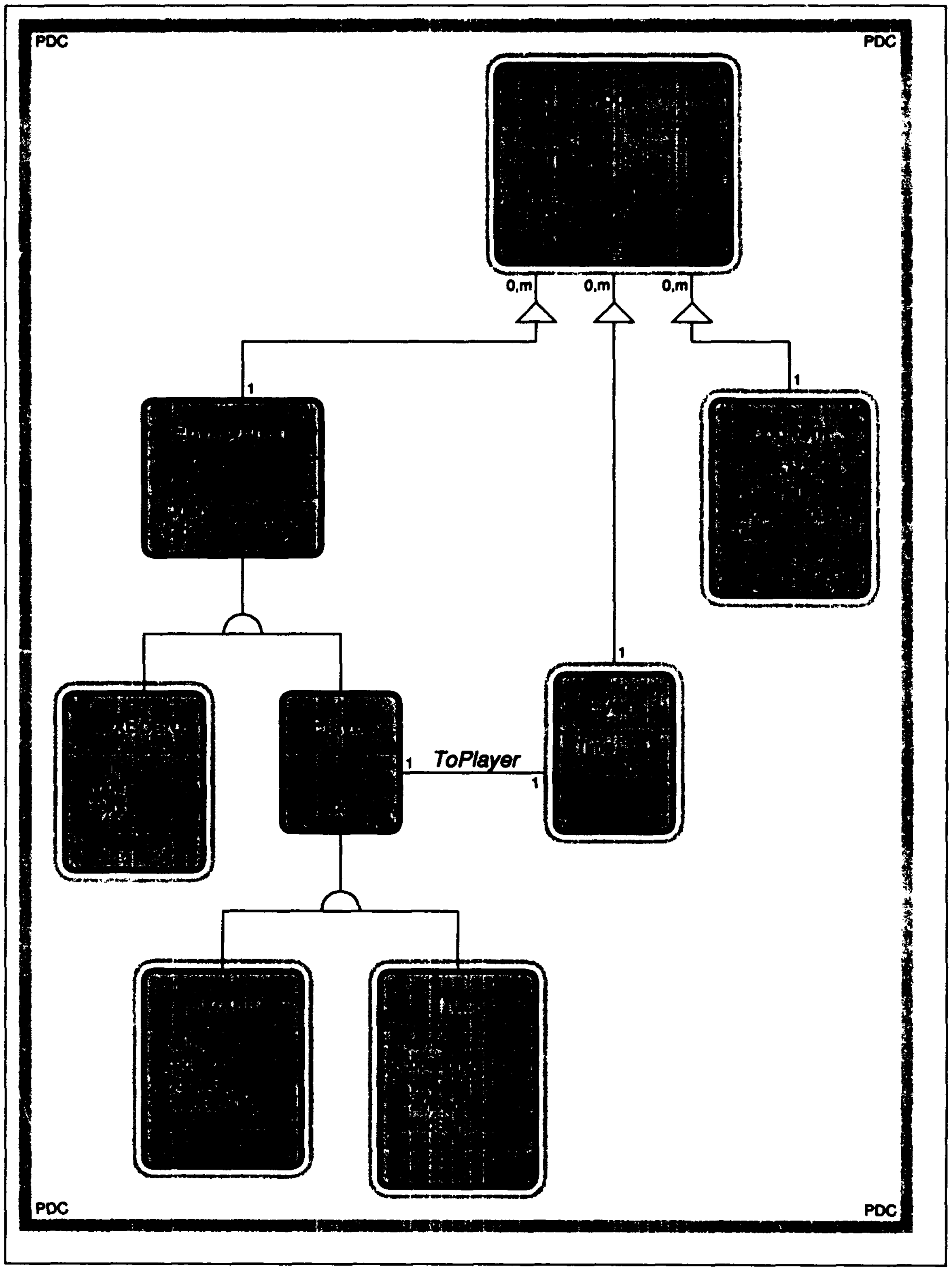

Figure 20. Simulation Benchmark Problem Domain OOD Diagram

a simulation object. Every subclass of the environment class must answer two queries, but most support more. The first required query is a query for the class name and the second is a query for the object instance's name. Simulation objects which are subclassed directly from the environment class are passive objects. For an active object to be created, it must be subclassed from the player class.

The *player* class is a virtual class which defines the additional services required by active simulation objects. The player class requires all its subclasses to define an *Execute* service. The execute service is called for an object when an event for that object is dispatched to it. Each event in the model contains a reference to the player object for which it is designated; this reference is labeled *ToPlayer* in Figure 20.

The *HexBoard* class is the only non-active simulation object in the benchmark model. This class contains three attributes: name, width, and height. The class allows queries on its width and height attributes in addition to the two basic queries required by the environment class.

The *Aircraft* and *Truck* classes are active simulation objects in the benchmark model. These classes are subclassed from the player class. These classes contain the attributes defined for them in the specification and allow queries on any of their attributes. The execute service of the aircraft class will accept *move* and *search* events, while the execute service of the truck class will accept only *move* events.

The *Event* class stores the future actions of all the simulation's active objects. Each event instance contains a single action for one object. For example, an event could tell aircraft `AIR-00345` to *search*. The event maintains a reference to the object, shown as *ToPlayer* in Figure 20. An event is dispatched to its object when it becomes the next time ordered event. Events are dispatched by the *RunUntilSimGoalTime* service of the model class. When an event is dispatched, the *Execute* service for the object it references is called with the event's *message* attribute passed as a parameter to the call.

The *LogItem* class is used to store a log of all the trucks which are found during aircraft searches. The simulation benchmark requires that this information be reported on the summary report which is generated from the post-processing portion of the simulation system. For each instance of the class, the information required by the summary report is included.

5.2.3 Design of the Database Interface. The database interface portion of our design interfaces the Motif user interface with the persistent objects which make up our simulation model. It is a good principle of user interface design to separate the interface from the application [17]. To accomplish this we developed an interface package which the user interface could call to perform actions on the object-oriented database. This avoided placing DBMS calls in the same module with Motif calls.

A possibly confusing portion of our design is the *simulation executive*. The simulation executive controls the execution of the simulation through time. In our design, the functions of the simulation executive are shared by the user interface and the model. The model provides the basic ability to simulate to a specified goal time, and the user interface controls more complex simulation executions, such as real time simulation. This was required due to the tight coupling of the user interface with simulation control. For example, the interface is required to display the current simulation time as each time slice is executed in the simulation. We leave it to future research to better integrate the simulation executive into a graphical simulation environment.

Our design for the simulation benchmark is now complete. Due to the nature of the benchmark, no design is necessary for the task management component. In the next section we describe our implementations of the simulation benchmark.

5.3 Simulation Benchmark Implementations

This section describes our implementations of the simulation benchmark. All of our implementations were built from the design described in the previous section. We created an implementation of the simulation benchmark for the ObjectStore DBMS, and we also created a non-persistent version of the benchmark. The non-persistent version does not provide most of the capability required by the benchmark but does provide a measure of how fast the simulation would execute if it was allowed to execute exclusively in memory.

A screen-print of our simulation benchmark implementation is shown in Figure 21. This figure shows the ObjectStore DBMS version of the benchmark.

5.3.1 ObjectStore Implementation. The first implementation of the simulation was done using the ObjectStore DBMS. With our experience developing the OO1 benchmark in ObjectStore, the implementation was reasonably simple. ObjectStore provided inheritance in the same manner as the C++ programming language and the hierarchy required by our simulation benchmark design was not difficult to implement.

5.3.2 Non-Persistent Implementation. The non-persistent implementation of the simulation benchmark was developed from the ObjectStore version of the benchmark. Due to the close ties between ObjectStore and the C++ language, we were able to remove all the database commands and execute the program as a stand alone C++ program. The non-persistent version of the benchmark is interesting because it represents the current typical method of simulation execution (where all the simulation data structures execute only in memory). The non-persistent version of the benchmark is not a valid implementation of the benchmark because it provides no support for multi-user use, nor for saving the simulation results. It does provide us with a yardstick to measure the performance cost for the additional functionality of an object-oriented DBMS.

5.3.3 Itasca and Matisse. We did not complete implementations of the simulation benchmark for Itasca or Matisse due to time constraints.

5.3.4 Multiple Object Model Problem. Using Motif in the C++ programming language along with an object-oriented DBMS required an understanding of several different object models. The C++ programming language provides an object model. The Xt toolkit (the basis for Motif programming) uses a different object model based upon the use of coding conventions. This plethora of object models adds unnecessary complexity to the task of program development and made our task of implementing the simulation benchmark more time consuming than anticipated.

[redacted]

[redacted]

Figure 21. The Simulation Benchmark Program (ObjectStore Version)

5.4 Summary

This chapter has described the simulation benchmark developed for this research. The specification for the simulation benchmark was described. Then our design for the benchmark was given. Finally, each implementation we developed for the benchmark was described. The next chapter, Chapter VI, examines our results from running the benchmarks.

VI. Benchmark Results Analysis

This chapter examines our results of the OO1 benchmark and the simulation benchmark. The configuration used for both benchmarks is shown in Table 4. For all our benchmark runs, two Sun SPARCstation 2 workstations were used. The first workstation acted as the database server and the other workstation as a client. On the server machine, the first hard disk held the operating system, the second held the database software, and the third held the benchmark databases. On the client workstation, the first hard disk held the operating system and the second held the ObjectStore client software. Neither Itasca nor Matisse required any software to be on the client machine (except the benchmark program). We first present our results for the OO1 benchmark and then our results for the simulation benchmark.

6.1 OO1 Benchmark Results

In this section we examine our results from the OO1 benchmark. The OO1 benchmark was implemented on the Itasca, Matisse, and ObjectStore object-oriented DBMSs as described in Chapter IV. We ran the benchmark in the following configurations:

- *Small remote* database
- *Small local* database
- *Small remote* database with no locality of reference
- *Small local* database with no locality of reference
- *Large remote* database
- *Large local* database

Only the *small remote* and the *large remote* configurations are required by the OO1 benchmark specification. The other configurations are optional [9].

For each benchmark configuration (such as the small local database configuration), five complete benchmark runs were made. The average of the five benchmark results is reported. Complete data for our OO1 benchmark results is contained in Appendix B and a statistical analysis of our results is presented in Appendix C.

Table 4. AFIT Benchmark System Configuration

Server Machine	*prowler*	Sun SPARCstation 2
	Operating System	Sun/OS 4.1.3
	Memory	48 Mbytes
	Swap	96 Mbytes
	Hard Disks	200 Mbyte
		200 Mbyte
		200 Mbyte
Client Machine	*doc*	Sun SPARCstation 2
	Operating System	Sun/OS 4.1.3
	Memory	48 Mbytes
	Swap	96 Mbytes
	Hard Disks	200 Mbyte
		200 Mbyte
DBMS Software	Itasca	Version 2.2
	Matisse	Version 2.2.0
	ObjectStore	Version 2.0.1

The next four sections describe our results for the small database configuration. The small database for the OO1 benchmark contains 20,000 parts and 60,000 connections between those parts. All the reported values for elapsed time in our results are in seconds.

6.1.1 Small Remote Database Results. The most important results for the OO1 benchmark are shown in Table 5, and graphically in Figure 22. The OO1 benchmark total is identified in Table 5 by *L+T+I*, which represents the sum of the lookup, traversal, and insert measurements. ObjectStore is clearly the best performer. The small remote database is the configuration which Cattell states most object-oriented DBMS applications require [9]. This configuration models a network client working with a database server located on a remote computer system. ObjectStore is 972% faster than Itasca and 243% faster than Matisse in our cold results. For the warm results, ObjectStore is 9551% faster than Itasca and 2391% faster than Matisse.[1] An analysis of our results, which is detailed in Appendix C, shows that they are statistically significant at the $\alpha = 0.05$ level.

[1]To calculate that database A is n% faster than database B, we used the following formula: $\frac{time_B}{time_A} = 1 + \frac{n}{100}$. This definition is documented by Hennessy and Patterson in [18].

Table 5. OO1 Benchmark Results for *Small Remote* Database

	Lookup		Traversal		Insert		L+T+I	
DBMS	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
Itasca	277.278	213.162	347.040	251.136	134.765	129.935	759.083	594.233
Matisse	125.982	68.448	53.111	35.387	64.076	49.507	243.169	153.382
ObjectStore	28.191	1.239	26.322	1.734	16.285	3.184	70.798	6.157

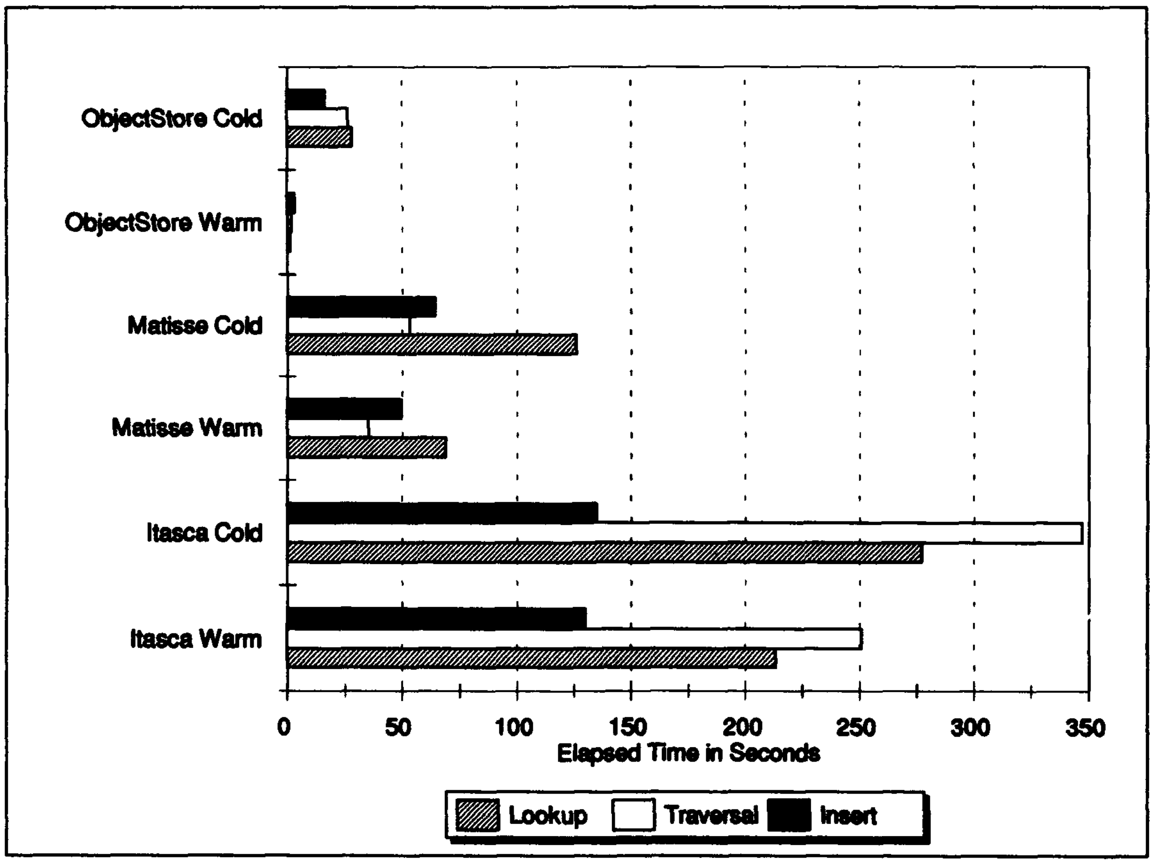

Figure 22. OO1 Benchmark Results for *Small Remote* Database

We were surprised at the large differences between the benchmark results. Cattell proposed a goal of about 10 seconds for each measurement (30 seconds for the L+T+I benchmark total) [9]. ObjectStore meets this objective for its warm results. The Itasca and Matisse results are much to slow to meet this goal even in the warm results, although Matisse is closer than Itasca.

There is a much larger change between the cold and warm results for ObjectStore than for Itasca and Matisse. From Table 5 we can calculate that the warm results are 28% faster than the cold results for Itasca, 59% faster than the cold results for Matisse, and 1050% faster than the cold results for ObjectStore. We believe that this is due to the virtual memory mapping supported by ObjectStore. Once data is read from the ObjectStore server it is accessed at memory speeds. Another reason for the difference is the benchmark database size. The benchmark database is smaller (see Figure 13) in ObjectStore than it is in Itasca or Matisse. If an application's database takes up less total storage then more of it can be cached in a fixed amount of memory.

The next section examines results for the small database size, but with the benchmark program run locally rather than remotely.

6.1.2 Small Local Database Results. After examining the results for the remote case, it is interesting to examine results for a local client. In the local configuration, the client executes on the same computer as the database server. Our results for this configuration are shown in Table 6, and graphically in Figure 23. Although we do see an improvement in the results for Itasca and Matisse, ObjectStore remains the clear winner. ObjectStore is 734% faster than Itasca and 199% faster than Matisse in our cold results. For the warm results, ObjectStore is 7797% faster than Itasca and 1784% faster than Matisse. The results are statistically significant at the $\alpha = 0.05$ level.

The performance of Itasca and Matisse improved in the local configuration. Itasca was 20% faster for the cold results and 12% faster for the warm results. Matisse was 7% faster for the cold results and 21% faster for the warm results.

In a very surprising result, ObjectStore was faster in the remote configuration than in the local configuration. This was true for all the OO1 benchmark measures. Our results

Table 6. OO1 Benchmark Results for *Small Local* Database

DBMS	Lookup Cold	Lookup Warm	Traversal Cold	Traversal Warm	Insert Cold	Insert Warm	L+T+I Cold	L+T+I Warm
Itasca	251.064	200.107	248.967	211.769	130.855	120.430	630.886	532.306
Matisse	91.652	36.058	70.499	41.030	64.279	49.899	226.431	126.987
ObjectStore	28.859	1.274	29.029	2.204	17.791	3.264	75.680	6.741

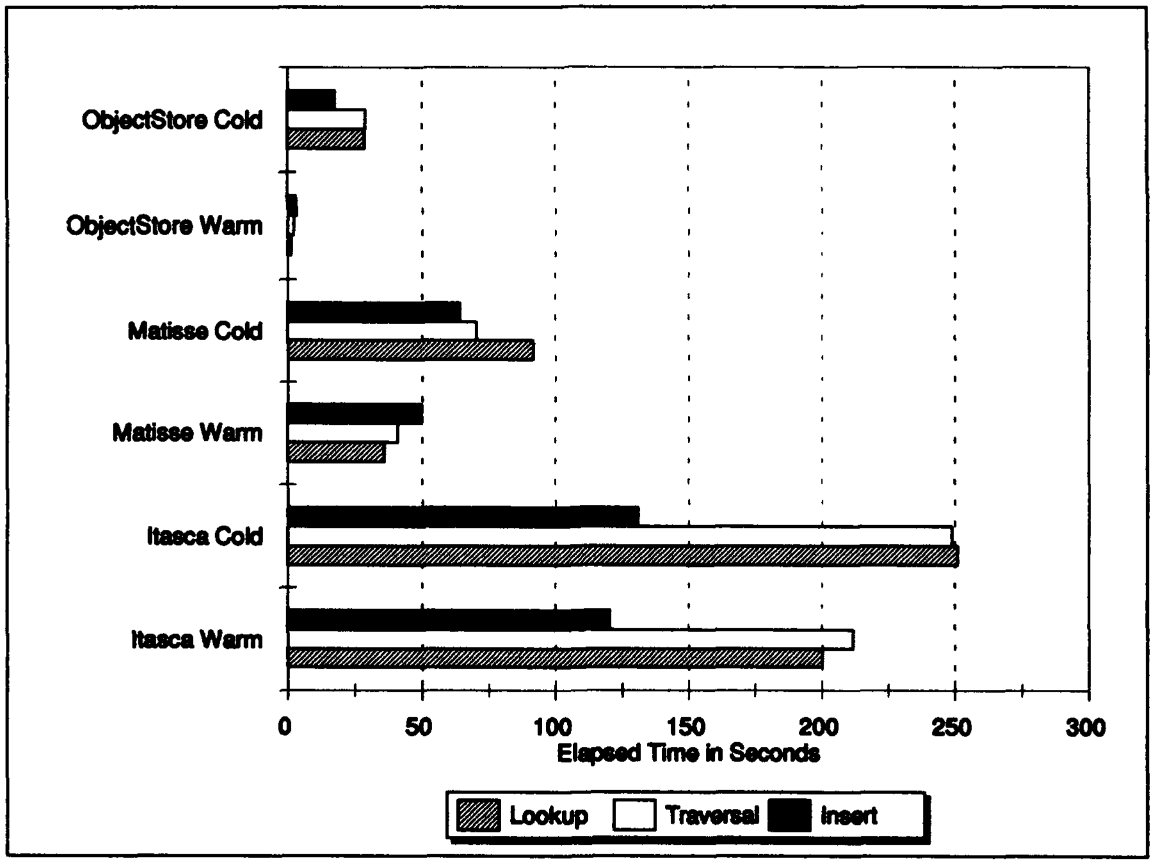

Figure 23. OO1 Benchmark Results for *Small Local* Database

Table 7. OO1 Benchmark Results for *Small Remote* Database (NLOR)

DBMS	Lookup Cold	Lookup Warm	Traversal Cold	Traversal Warm	Insert Cold	Insert Warm	L+T+I Cold	L+T+I Warm
Itasca	252.481	203.955	380.217	301.063	206.002	160.953	838.700	665.972
Matisse	146.186	78.955	120.816	63.672	88.340	61.556	355.342	204.182
ObjectStore	29.765	1.250	39.574	1.766	33.045	9.919	102.384	12.936

show the remote configuration was 6.9% faster than the local configuration for the cold time and 9.49% faster than the local configuration for the warm time. However, when we examined these results statistically (see Appendix C), we found that there is not enough evidence to prove that this finding is significant at the $\alpha = 0.05$ level. It is possible that competition between the client program (our benchmark) and the database server for the same computer's resources may have caused this result.

6.1.3 Small Remote Database (NLOR) Results. The OO1 benchmark requires that the database be built with a large degree of *locality of reference*. 90% of the connections between parts must be randomly connected to 1% of the closest parts. Parts are close if they have numerically similar part identifiers [9]. To further investigate the performance of our three databases, we removed this requirement. Our results for the small local database with no locality of reference (NLOR) database configuration are shown in Table 7, and graphically in Figure 24. ObjectStore is still the top performer, even without the locality of reference requirement. ObjectStore is 719% faster than Itasca and 247% faster than Matisse in our cold results. For the warm results, ObjectStore is 5048% faster than Itasca and 1478% faster than Matisse. The results are statistically significant at the $\alpha = 0.05$ level.

ObjectStore, in the warm results, is the most affected by the loss of locality of reference. ObjectStore is 110% faster with locality of reference, Itasca is 12% faster with locality of reference, and Matisse is 33% faster with locality of reference. For the cold results, ObjectStore is 45% faster with locality of reference, Itasca is 10% faster with

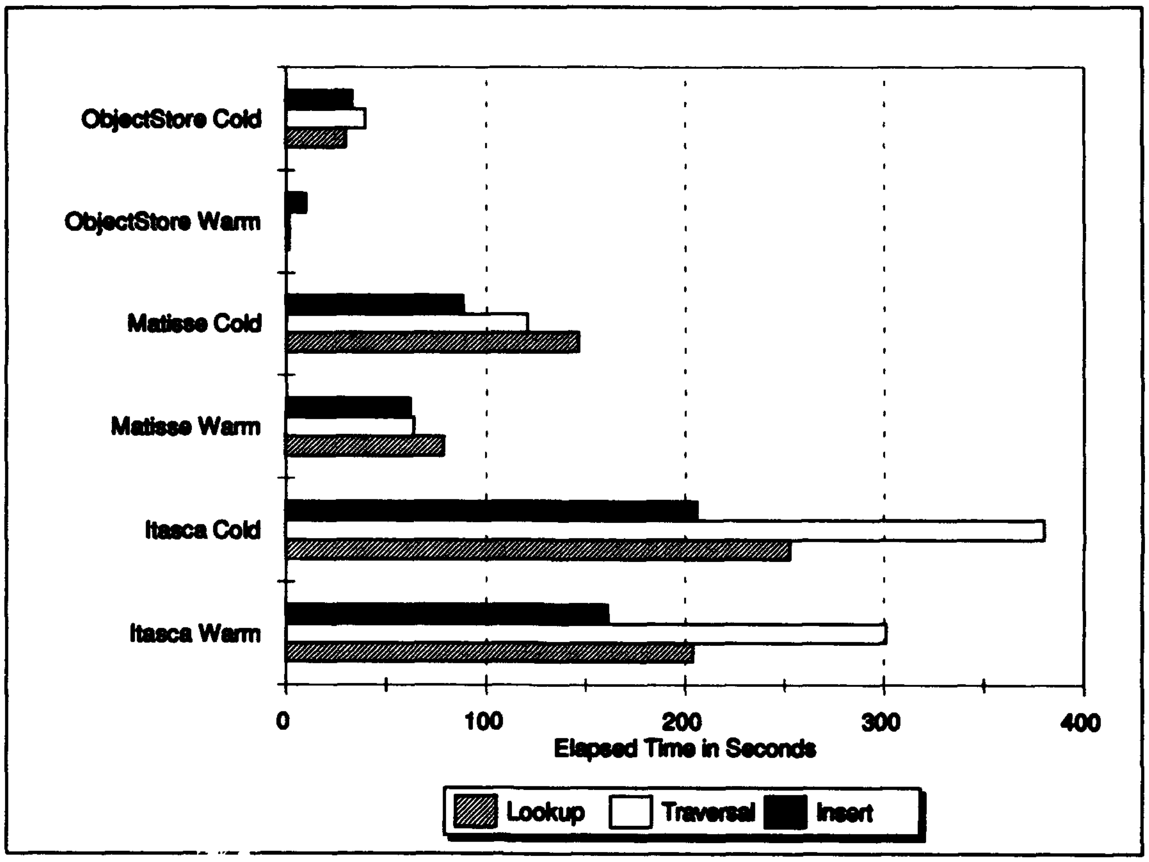

Figure 24. OO1 Benchmark Results for *Small Remote* Database (NLOR)

Table 8. OO1 Benchmark Results for *Small Local* Database (NLOR)

	Lookup		Traversal		Insert		L+T+I	
DBMS	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
Itasca	234.236	197.707	342.590	270.023	188.591	150.635	765.417	618.365
Matisse	107.941	31.779	145.648	58.183	98.353	61.876	351.942	151.839
ObjectStore	33.249	1.312	44.217	1.792	39.102	11.914	116.568	15.017

locality of reference, and Matisse is 46% faster with locality of reference. Itasca is affected the least by the loss of locality of reference.

6.1.4 Small Local Database (NLOR) Results. Our results for the small local database with no locality of reference database configuration are shown in Table 8, and graphically in Figure 25. ObjectStore is the top performer. ObjectStore is 557% faster than Itasca and 202% faster than Matisse in our cold results. For the warm results, ObjectStore is 4018% faster than Itasca and 911% faster than Matisse.

We experienced problems with Itasca in this benchmark configuration. The database could not get enough system memory to execute the benchmark properly. We were only able to complete a single benchmark run and obtain partial results for the other four runs (see Appendix B). Itasca support was unable to help us solve this problem since it was a limitation of our workstations. Hence, while we were able to show statistical significance for the differences in the Matisse and ObjectStore results to the $\alpha = 0.05$ level, this was not possible for Itasca.

This concludes our results for the small database configuration. A summary of our OO1 benchmark results is shown in Figure 9 for the small database configurations. Figure 9 shows all our results in terms of what percentage ObjectStore is faster than Itasca or Matisse. The next two sections describe our results for the large database configuration. The large database contains 200,000 parts and 600,000 connections between those parts.

6.1.5 Large Remote Database Results. Our results for the large database configurations are not as complete as those obtained for the small database configurations.

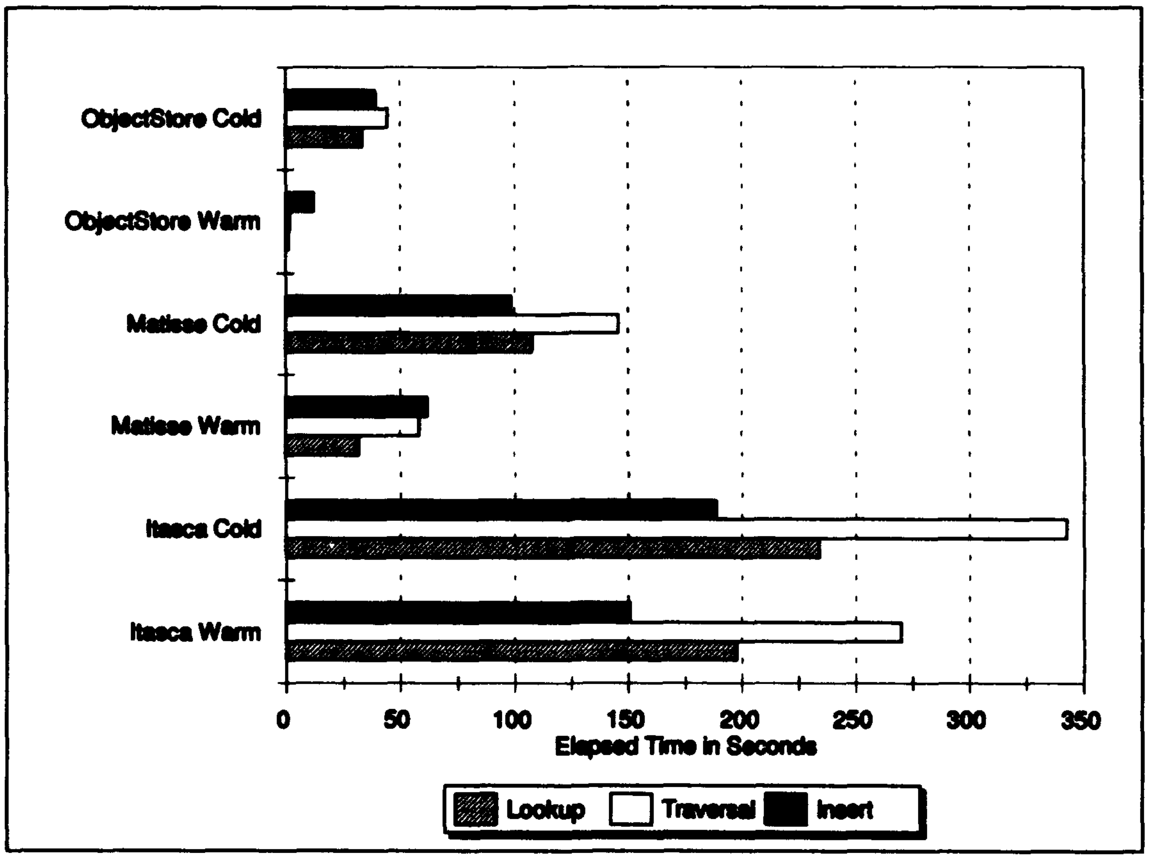

Figure 25. OO1 Benchmark Results for *Small Local* Database (NLOR)

Table 9. Summary of OO1 Benchmark *Small* Database Results

Benchmark Configuration			ObjectStore versus Itasca	ObjectStore versus Matisse
Small Remote	cold	Lookup	884% faster	347% faster
		Traversal	1218% faster	102% faster
		Insert	728% faster	293% faster
		L+T+I	972% faster	243% faster
	warm	Lookup	17104% faster	5428% faster
		Traversal	14383% faster	1941% faster
		Insert	3981% faster	1455% faster
		L+T+I	9551% faster	2391% faster
Small Local	cold	Lookup	770% faster	218% faster
		Traversal	758% faster	143% faster
		Insert	636% faster	261% faster
		L+T+I	734% faster	199% faster
	warm	Lookup	15607% faster	2730% faster
		Traversal	9508% faster	1762% faster
		Insert	3590% faster	1429% faster
		L+T+I	7797% faster	1784% faster
Small Remote (NLOR)	cold	Lookup	748% faster	391% faster
		Traversal	861% faster	205% faster
		Insert	523% faster	167% faster
		L+T+I	719% faster	247% faster
	warm	Lookup	16216% faster	6216% faster
		Traversal	16948% faster	3505% faster
		Insert	1523% faster	521% faster
		L+T+I	5048% faster	1478% faster
Small Local (NLOR)	cold	Lookup	604% faster	225% faster
		Traversal	675% faster	229% faster
		Insert	382% faster	151% faster
		L+T+I	557% faster	202% faster
	warm	Lookup	14969% faster	2322% faster
		Traversal	14968% faster	3147% faster
		Insert	1164% faster	419% faster
		L+T+I	4018% faster	911% faster

We encountered problems with Itasca and Matisse when attempting to build the large benchmark databases and were unable to complete large database measurements on the Itasca DBMS.

Itasca, as was seen in the small local NLOR configuration, did not have enough memory to load the large database. The Itasca database server would consume system memory until the benchmark program, which was using memory from the same system, would fail. In the attempts we made to build the large database in Itasca, failure would occur after the program was allowed to run for about 27 hours. We estimated that a remote build of the large database would have taken about 10 days but did not attempt such a build due to the low probability of success.

We encountered a very different problem with the Matisse DBMS. Matisse writes data to the disk without attempting to compact the data into the smallest space. Therefore, large amounts of disk space are consumed during the creation of the OO1 benchmark database. To compact the disk space, a program called *mts_collect_versions* must be executed on the database. For the small database configuration, we allowed the database to be built and then ran the collect versions program. But for the large database, we did not have enough disk space to follow this procedure. We worked with Matisse support to come up with a solution and decided the only solution was to manually monitor the database load. For the load, we allocated the entire 200 Mbytes of our database hard disk to Matisse. The load was run until 90% of the Matisse database was filled, then the load program was stopped (using the control-Z command). With the load stopped, the collect versions program was executed to compact the disk space in use by Matisse. Using this method we were able to build the large benchmark database in Matisse in about 1.5 weeks. We were then able to execute the OO1 benchmark measures.

We encountered no problems with the ObjectStore database for the large database measures. We examine the remote results in this section and the local results in the next.

Our results for the large remote database configuration are shown in Table 10, and graphically in Figure 26. ObjectStore is the best performer, but not by as wide a margin as in the small database configuration. ObjectStore is 37% faster than Matisse in our cold

Table 10. OO1 Benchmark Results for *Large Remote* Database

DBMS	Lookup		Traversal		Insert		L+T+I	
	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
Matisse	245.811	216.350	231.703	214.405	119.756	110.767	577.456	535.937
ObjectStore	121.811	64.019	194.698	118.016	104.126	61.330	420.636	243.364

Table 11. OO1 Benchmark Results for *Large Local* Database

DBMS	Lookup		Traversal		Insert		L+T+I	
	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
Matisse	228.080	215.053	229.644	228.602	106.308	112.559	546.682	552.772
ObjectStore	105.463	56.753	183.473	155.877	70.533	45.905	359.469	258.535

results and 120% faster than Matisse in our warm results. The results are statistically significant at the $\alpha = 0.05$ level.

For ObjectStore the warm results are only 73% faster than the cold results. This difference is much less than the 1050% improvement seen in the small database results. For Matisse the warm results are only 8% faster than the cold results. Again, this difference is much less than the 59% improvement seen in the small database results. We believe that this difference is due to the larger size of the database *working set*. In the small database configuration ObjectStore must have been able to hold most (or all) of the benchmark's working set in the database cache, but was not able to do this for the large database configuration.

6.1.6 Large Local Database Results. Our results for the large local database configuration are shown in Table 11, and graphically in Figure 27. ObjectStore is again the best performer. ObjectStore is 52% faster than Matisse in our cold results and 114% faster than Matisse in our warm results. The results are statistically significant at the $\alpha = 0.05$ level.

As in the large remote configuration results, we see a much smaller difference between the cold and warm results than for the small database results. For example, the warm

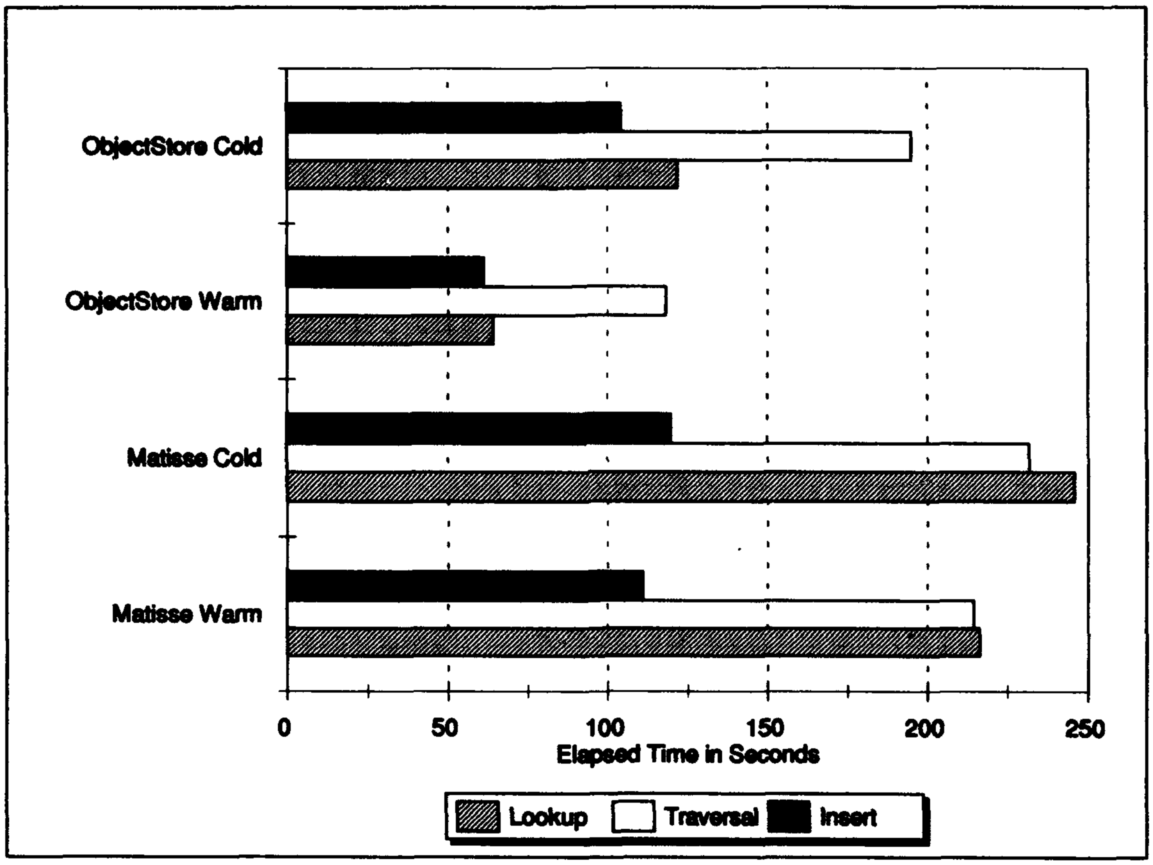

Figure 26. OO1 Benchmark Results for *Large Remote* Database

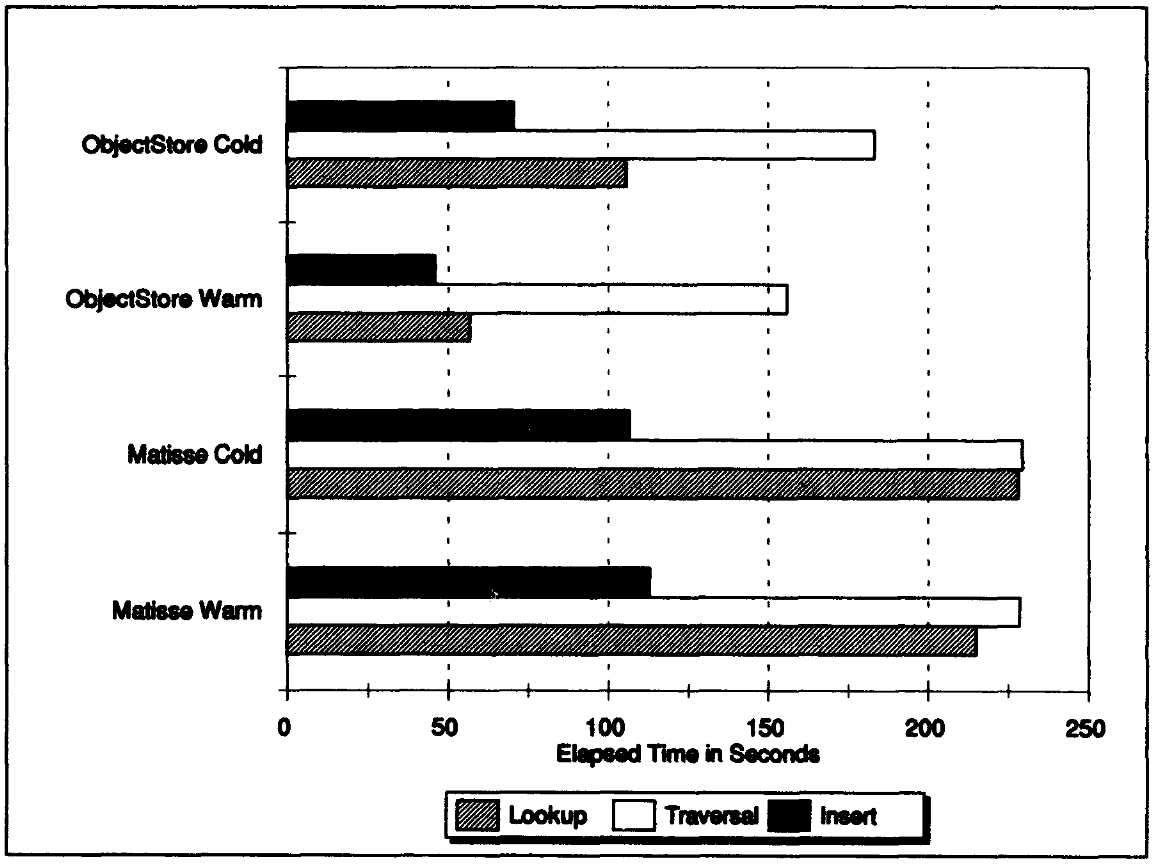

Figure 27. OO1 Benchmark Results for *Large Local* Database

results for ObjectStore are 39% faster than the cold results. For the same configuration of the small database they were 1103% faster.

A summary of our OO1 benchmark results is shown in Figure 12 for the large database configurations. Figure 12 shows all our results in terms of what percentage ObjectStore is faster than Matisse.

6.1.7 Benchmark Database Load Times. This section examines the time required to build the OO1 benchmark databases. The build times and database sizes for the small database are shown in Table 13. Note that for our three DBMSs, there is a considerable difference in both the amount of time required and the amount of disk space necessary. The Matisse database required a much larger amount of disk space during the build, but was compacted down to the size shown in Table 13 before the benchmark runs were made. We noted the large amount of space overhead required by Itasca and Matisse. Cattell estimated a size between 4 and 5 Mbytes for the small database. Only ObjectStore was within this range [9].

The build times and database sizes for the large database are shown in Table 14.

6.1.8 Matisse Version Collection Results. To determine the effect of version collection on the Matisse DBMS, we varied the delay allowed for version collection between benchmark runs. For all our benchmark configurations using Matisse, we started the version collection program after a run. Then, we immediately started the next run. However, for the *Small Local* database configuration and the *Small Remote* (NLOR) database configuration, we also tested with a 200 second delay to allow version collection. The results, which were inconclusive, make it apparent that the version collection program does not have priority over an application program (see Appendix B). Matisse support confirmed this presumption by informing us that the version collection program runs with a very low priority inside the Matisse server.

6.1.9 Reverse Traversal Results. The OO1 benchmark requires that a reverse traversal be run as one of the measures. We indeed executed the reverse traversal on all our configurations but found that the results varied tremendously. This variation was due to

Table 12. Summary of OO1 Benchmark *Large* Database Results

Benchmark Configuration			ObjectStore versus Matisse
Large Remote	cold	Lookup	102% faster
		Traversal	19% faster
		Insert	15% faster
		L+T+I	37% faster
	warm	Lookup	238% faster
		Traversal	82% faster
		Insert	81% faster
		L+T+I	120% faster
Large Local	cold	Lookup	116% faster
		Traversal	25% faster
		Insert	51% faster
		L+T+I	52% faster
	warm	Lookup	279% faster
		Traversal	47% faster
		Insert	145% faster
		L+T+I	114% faster

Table 13. OO1 Benchmark Load Times for *Small* Database

	Itasca	Matisse	ObjectStore
Elapsed Time	61738.459	9963.483	102.481
Database Size (in Kbytes)	12470.956	17338.000	4664.000

Table 14. OO1 Benchmark Load Times for *Large* Database

	Matisse	ObjectStore
Elapsed Time	~1.5 Weeks	23.130 Hours
Database Size (in Kbytes)	171400.000	45296.000

Table 15. Summary of OO1 Benchmark Results

Benchmark Configuration			ObjectStore versus	
			Itasca	Matisse
Small Remote	cold	L+T+I	972% faster	243% faster
	warm	L+T+I	9551% faster	2391% faster
Small Local	cold	L+T+I	734% faster	199% faster
	warm	L+T+I	7797% faster	1784% faster
Small Remote	cold	L+T+I	719% faster	247% faster
	warm	L+T+I	5048% faster	1478% faster
Small Local	cold	L+T+I	557% faster	202% faster
	warm	L+T+I	4018% faster	911% faster
Large Remote	cold	L+T+I		37% faster
	warm	L+T+I		120% faster
Large Local	cold	L+T+I		52% faster
	warm	L+T+I		114% faster

the random nature of the number of parts found in a reverse traversal. We have reported the complete results for the reverse traversal in Appendix B but have not included them in this chapter because there was no statistical significance to the results. The benchmark specification recognizes the problems with the reverse traversal measure and did not include it in the benchmark total.

This concludes our results for the OO1 benchmark. A summary of our OO1 benchmark results is shown in Figure 15. In the next section we present our results for the simulation benchmark.

6.2 Simulation Benchmark Results

This section examines our simulation benchmark results. The simulation benchmark was described in Chapter V. Two implementations of the simulation benchmark have been completed to date: a persistent version using the ObjectStore DBMS and a non-persistent version created in the C++ programming language.

The non-persistent version of the benchmark is not a valid implementation of the benchmark because it provides no transaction model and is not persistent (it can not save any data to disk). What the implementation does show is the performance which would be obtained by most current simulation systems (which run exclusively in memory). The performance difference between the non-persistent version of the benchmark simulation and the ObjectStore version provide a yardstick to measure the performance price paid for the functionality gains provided by the object-oriented DBMS.

The next two sections describe the quantitative and the qualitative results for the simulation benchmark. We only present results for the small benchmark database. Measurements for the large database configuration were not done due to time constraints.

6.2.1 Quantitative Results. For each benchmark configuration five complete benchmark runs were made. The average of the five benchmark measures is reported. Complete data for our simulation benchmark results is contained in Appendix D and a statistical analysis of our results is presented in Appendix E. A summary of our results is shown in Table 16.

The ObjectStore version of the benchmark performs much better than we expected compared to the non-persistent version. For the ObjectStore version of the benchmark, the time slice defines the size of a transaction in terms of simulation time. It can be seen from the *hour run* results in Table 16 that as the time slice is increased, the performance loss due to use of the object-oriented DBMS decreases until it becomes almost negligible. Note that the non-persistent version of the benchmark does not write any of the model data to the disk, so additional time would be required to save the model data in an actual simulation system. A graph of the *hour run* results by the time slice setting is shown in Figure 28.

The ObjectStore results for the two post-processing measures are actually faster than the non-persistent version of the benchmark. This is not surprising because the ObjectStore DBMS did not have to write any data during these measures (they are both read-only transactions on the database), and all the model data should have been in the DBMS's cache.

6.2.2 Qualitative Results. Overall we found the ObjectStore DBMS to provide a good platform for simulation system development. The DBMS provided a large amount

Table 16. Simulation Benchmark Results for *Small* Database

Benchmark Measure	ObjectStore (*Remote*)	ObjectStore (*Local*)	Non-Persistent
Model Creation	0.645	0.633	0.007
Scenario Creation	4.184	4.285	1.603
Hour Run (TS = 60)	576.894	580.294	392.738
Hour Run (TS = 600)	485.341	490.979	389.576
Hour Run (TS = 1800)	476.418	482.764	388.308
Hour Run (TS = 3600)	474.981	477.213	387.685
Simulation Throughput	0.159	0.180	0.115
Version Creation	20.794	21.168	14.138
Map Creation	9.805	10.243	10.718
Report Creation	2.613	2.999	2.336

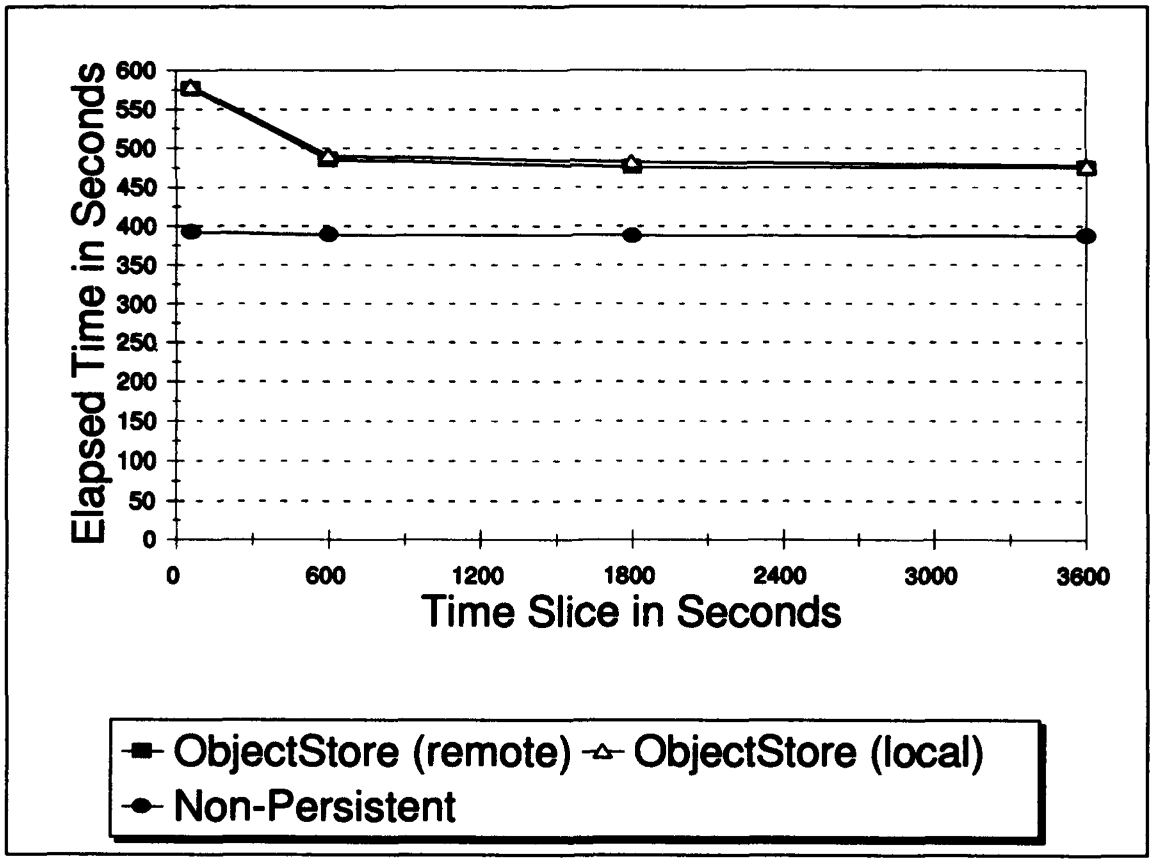

Figure 28. Simulation Benchmark Hour Run Results by Time Slice Setting for *Small* Database

of functionality with a minimum of performance overhead. The following is a list of some of the functional benefits provided by the ObjectStore DBMS to simulation systems:

- *Similarity to the C++ Programming Language*: Due to ObjectStore's close ties to the C++ programming language, the ObjectStore version of the simulation benchmark looks very much like a C++ program. The transaction model and the declaration of persistent objects are the only major differences. This is only a benefit if a simulation system is being developed in C++. It is a drawback if another language, such as Ada, is being used for development.
- *Motif Interface*: The ObjectStore DBMS interacted very well with the Motif user interface developed for the simulation benchmark.
- *Multi-User Access to Model Data*: ObjectStore controlled all concurrent access to the database. When two executing versions of the simulation benchmark worked with the same model, we encountered no consistency problems. For example, we were able to execute the simulation on one workstation and update a map on a remote workstation. To provide this type of multi-user concurrency control without the object-oriented DBMS would have required a large amount of code (the non-persistent version of the benchmark did not allow multi-user access). Multi-user access is a major benefit of object-oriented DBMS use.
- *Browser Tool Use*: The ObjectStore DBMS provides a graphical database browser tool. We used this tool to examine the simulation benchmark database in an ad-hoc fashion. We believe that graphical browser tools are useful for simulation systems.

6.3 Summary

This chapter has presented our results for the OO1 benchmark and the simulation benchmark. Chapter VII presents conclusions and recommendations based upon our benchmark results.

VII. Conclusions and Recommendations

7.1 Overview

In this thesis we studied the performance of the Itasca, Matisse, and ObjectStore object-oriented DBMSs. The OO1 benchmark was developed and run on the three DBMSs. A new benchmark, the AFIT Simulation benchmark, was developed and implemented on the ObjectStore DBMS. In this chapter, we present our conclusions based upon the results presented in Chapter VI and summarize some important lessons learned about benchmarking. Finally, recommendations are presented for future research in object-oriented DBMS performance.

7.2 Conclusions

This section presents our conclusions. These conclusions are based upon our work with the three commercial object-oriented DBMS and our benchmark results which were presented in Chapter VI.

- ObjectStore was the top performer on the OO1 benchmark. In the most critical benchmark configuration, the *small remote* database, ObjectStore was the clear winner. ObjectStore was 972% faster than Itasca and 243% faster than Matisse in our cold results, and was 9551% faster than Itasca and 2391% faster than Matisse in our warm results. In all the other configurations we tested, ObjectStore was the fastest DBMS, although it was found that ObjectStore was the most sensitive to locality of reference.
- There is *wide* variation in the performance of commercial object-oriented DBMSs. Investigating the performance of an object-oriented DBMS is critical. The commercial systems available today are by no means a commodity item (as relational systems are becoming). An object-oriented DBMS may have all the functional capability required by an application, but its performance may be too slow.
- A programming language interface to an object-oriented DBMS should be closely tied to a specific language or not tied to any language at all. It proved to be very confusing

to work with the Itasca C++ API which provided a blend of C++ functionality with DBMSs functionality. We did not examine the Itasca Lisp API during this research. The ObjectStore programming interface, which was closely tied to the C++ programming language, and the Matisse programming interface, which was not closely tied to a specific language, were much easier to learn and work with.

- Some object-oriented DBMSs provide sufficient performance to allow the creation of persistent simulation environments. Our results on the simulation benchmark show that for some areas of simulation, the loss of performance is acceptable given the large gains in functionality. This is especially true if concurrent access to executing simulation data is required, such as animating a running simulation.

7.3 Lessons Learned

The following is a summary of the valuable lessons learned during our benchmarking effort:

- *Use Vendor Customer Support*: Without the aid of the support groups from Itasca, Matisse, and ObjectStore, it is unlikely we would have solved many of the problems we encountered. The current implementations of object-oriented DBMSs are very complex pieces of system software which are difficult to understand, especially when working with several of them at the same time. The aid of vendor customer support is essential to any benchmarking effort.

  We recommend sending benchmark source code through vendor customer support several times because they are often very busy and will not always examine the details of your implementation in just one look.

- *Plan for Large Amounts of Data*: We underestimated the large amount of data which would be generated during this research effort. The data took a considerable amount of time to consolidate and present in an understandable format. Some up-front planning of the steps necessary to move benchmark data from the program output to the final report will save a large amount of effort and confusion. We also recommend that the process be automated to avoid transcription errors. For this research effort, custom C programs were used to filter the benchmark output into a spreadsheet [5] for calculation. Then, a second custom C program was used to filter the calculated results into TEX for our final report.

- *Limit Source Code Tuning*: There is a point where making minor changes to the benchmark source code to improve the performance is not effective. A large amount of time can be spent working on the source code of benchmark implementations. We recommend planning a final stop-work date for all the benchmark implementations and working with the vendors to get the best product out by that time.
- *Disk Space*: One of the recurring problems during our research was that of the database disk filling up. We underestimated the amount of storage necessary to work with three DBMS systems at the same time. We recommend determining how much space will be necessary to hold the largest database required for a benchmark, then doubling that size. The additional space may be used for backups, or may be needed if one of the DBMSs requires more disk space than anticipated.
- *Coding Standards*: We developed a set of coding standards at the very start of our research. These standards helped provide consistency across all our implementations. Consistency can help to ensure that implementations on different DBMSs are done fairly.

7.4 Recommendations

The following are recommendations for areas of further research which we feel are needed in object-oriented DBMS performance and simulation:

- A larger simulation system should be built using the ObjectStore DBMS. ObjectStore consistently performed much better than Itasca or Matisse in the OO1 benchmark and added very little performance overhead compared to the non-persistent version in the simulation benchmark. ObjectStore should be investigated for use in an actual simulation system.
- The simulation benchmark should be extended to test the use of long transactions and version management. The current ObjectStore implementation of the simulation

benchmark is set up to allow multiple versions of a model, but the user interface does not provide an interface to this functionality. This was partially due to time constraints and partially due to a lack of research into how a version control system can be used by a simulation system. It is possible that the version management system provided by object-oriented DBMSs to support long transactions could be useful in the simulation domain or it may be that some changes to the version management model (which was developed for engineering applications such as CAD and CASE) need to be made.

7.5 Summary

Our OO1 benchmark results for three commercial object-oriented DBMSs show that users must be very wary when planning to use an object-oriented DBMS for any application. Even though the DBMS may provide the functional capabilities required, the system may not provide the level of performance required. Although our simulation benchmark showed benefits from the use of an object-oriented DBMS in the simulation domain, further investigation is necessary, especially in the area of version management.

Appendix A. Object-Oriented Analysis and Design Notation Summary

This appendix summarizes the object-oriented analysis and design notations used during this research. Figure 29 gives a summary of the Coad/Yourdon object-oriented analysis (OOA) and object-oriented design (OOD) notations. This notation was presented in the books *Object-Oriented Analysis*, and *Object-Oriented Design* [10, 11].

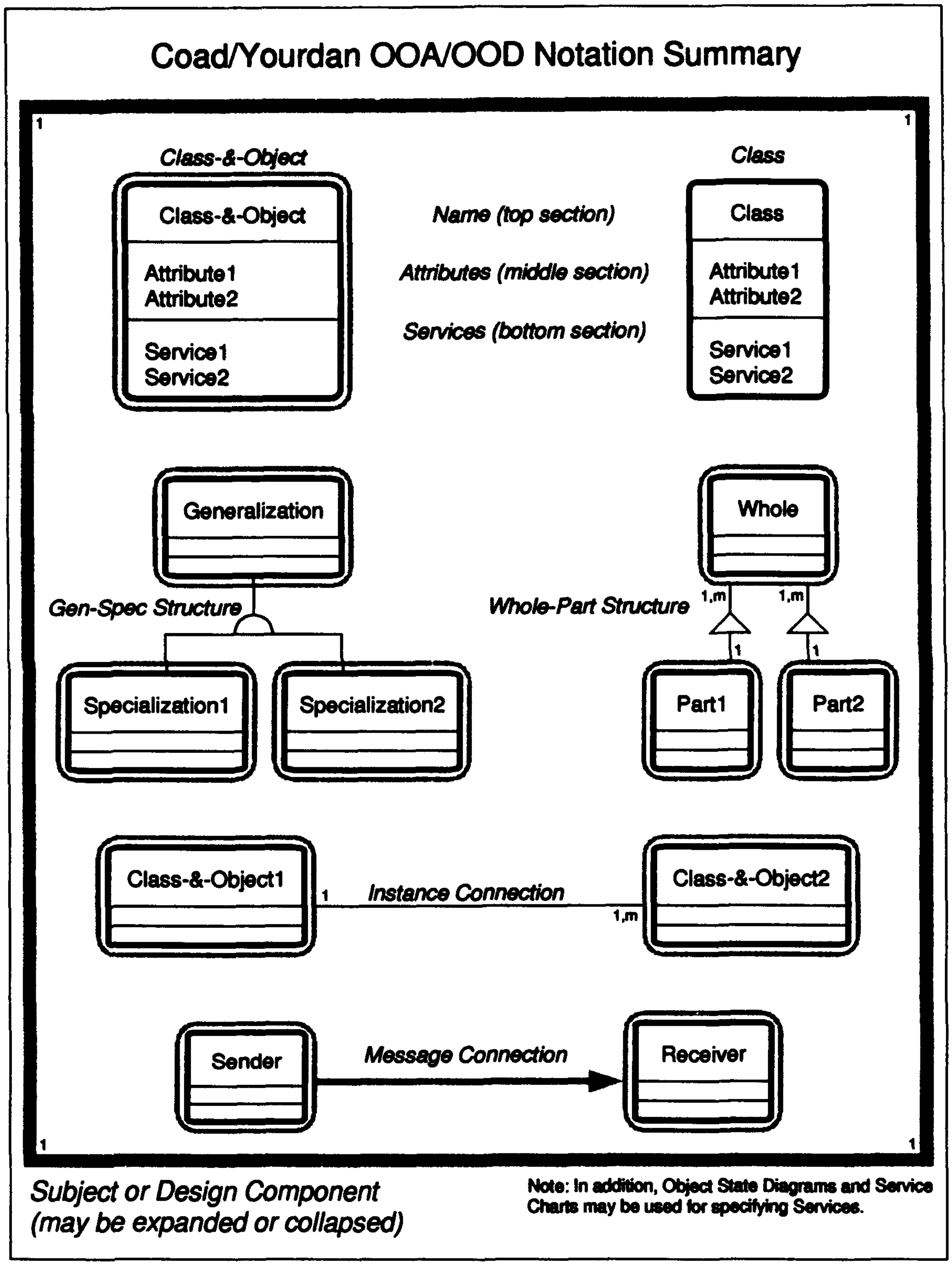

Figure 29. Coad/Yourdan OOA/OOD Notations

Appendix B. Detailed OO1 Benchmark Results

This appendix contains, in detail, the results obtained from our runs of the OO1 benchmark. Benchmark results are included for the Itasca, Matisse, and ObjectStore object-oriented DBMSs.

For each different benchmark configuration (object-oriented DBMS, database size, and benchmark variation) five complete runs of the OO1 benchmark were done. Five runs were made rather than one to try to obtain a better picture of typical DBMS performance, not just a single snapshot. This appendix contains both raw and summary results for all benchmark configurations. Benchmark results for each configuration are reported using three tables and two charts. The tables and charts are described below in the order which they appear for each benchmark configuration.

- *Summary Results Table*: This table reports the cold and warm times for the lookup, traversal, and insert measures of the benchmark, and the benchmark total (L+T+I). These values are reported for the five complete benchmark runs. The average and sample standard deviation of the five runs is given at the bottom of the table. All the times reported in this table are in seconds.
- *Benchmark Results Chart*: This bar chart provides a clear picture of the average benchmark results (recorded in the second from last line of the summary results table). It provides a graphical picture of the percentage of time spent in each individual measurement compared to the benchmark total (L+T+I). It is important to note that the y-axis scale of this chart is different for each benchmark configuration, so care must be taken when using this chart for comparisons.
- *Normalized Reverse Traversal Results Table*: This table reports the reverse traversal results normalized so that they can be compared to the forward traversal results. The layout of this table is the same as the summary results table. We decided to separate the reverse traversal results into a separate table because they are not included in the benchmark total (L+T+I). The formula for normalizing the reverse traversal results was described in the OO1 benchmark specification [9]. The normalization formula, where $T_{rt\ normalized}$ represents the normalized reverse traversal result, is shown in

Table 17. OO1 Benchmark Load Times for *Small* Database

	Itasca	Matisse	ObjectStore
Elapsed Time	61738.459	9963.483	102.481
Database Size (in Kbytes)	12470.956	17338.000	4664.000

Table 18. OO1 Benchmark Load Times for *Large* Database

	Matisse	ObjectStore
Elapsed Time	~1.5 Weeks	83266.697
Database Size (in Kbytes)	171400.000	45296.000

Equation 1 below.

$$T_{rt\ normalized} = T_{rt}\frac{N_{ft}}{N_{rt}} \tag{1}$$

In Equation 1 T_{rt} is the elapsed time measured and N_{rt} is the number of parts actually found in a single reverse traversal measure. N_{ft} is the number of parts found in a single forward traversal measure, which for the OO1 benchmark is always 3,280 parts. All the times reported in this table are in seconds.

- *Raw Benchmark Results Table*: This table reports the raw results for the five benchmark runs. In fact, this is the exact output given by our benchmark program and was automatically filtered into the form presented in this appendix to avoid transcription errors. All the times reported in this table are in seconds.

- *Average Individual Benchmark Measures Chart*: The OO1 benchmark requires that each measurement be performed 10 times for each complete iteration of the benchmark. These 10 runs are labeled *Run 1* through *Run 10* in this chart and in the raw benchmark results table. Run 1 is reported as the cold result and the average of runs 2 through 10 is reported as the warm result. This chart conveys two things. First, the L+T+I line displays the average total for each of the 10 runs. This line gives an graphical picture of the the average benchmark total time throughout the 10 runs. Second, the three bars displayed for each run provide a graphical view of the relationship between the individual benchmark measures to the L+T+I line throughout the 10 runs. It is important to note that the y-axis scale of this chart is different for each benchmark configuration, so care must be taken when using this chart for comparisons.

The database load times and the size required for the OO1 benchmark database are shown in Figure 17 for the small database and Figure 18 for the large database. The large database build for Matisse could not be automated (due to a lack of disk space) and was carried out in about 1.5 weeks.

B.1 OO1 Benchmark Results for the Itasca DBMS

This section reports our OO1 benchmark results for the Itasca DBMS. Results for the following database configurations are provided:

- Itasca *Small Remote* Database — The results for this benchmark configuration are reported in Tables 19, 20, and 21 and in Figures 30 and 31.
- Itasca *Small Local* Database — The results for this benchmark configuration are reported in Tables 22, 23, and 24 and in Figures 32 and 33.
- Itasca *Small Remote* Database with No Locality of Reference (NLOR) — The results for this benchmark configuration are reported in Tables 25, 26, and 27 and in Figures 34 and 35.
- Itasca *Small Local* Database with No Locality of Reference — The results for this benchmark configuration are reported in Tables 28 and 29 and in Figures 36 and 37. The results for this benchmark configuration are limited because the Itasca DBMS did not have enough memory to execute in this configuration. Only partial benchmark runs completed and no reverse traversal measures ran to completion.

Table 19. Itasca OO1 Summary Results for *Small Remote* Database

	Lookup		Traversal		Insert		L+T+I	
Benchmark	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	236.436	193.556	259.067	213.509	135.861	132.372	631.364	539.437
2	237.550	195.475	311.968	219.142	135.286	132.297	684.804	546.914
3	241.633	218.644	265.949	257.307	131.300	128.386	638.882	604.337
4	248.285	213.718	404.054	289.713	136.238	127.271	788.577	630.702
5	422.488	244.417	494.163	276.011	135.139	129.348	1051.790	649.776
Average:	277.278	213.162	347.040	251.136	134.765	129.935	759.083	594.233
Sample STD:	81.307	20.643	100.550	33.854	1.987	2.311	175.232	49.393

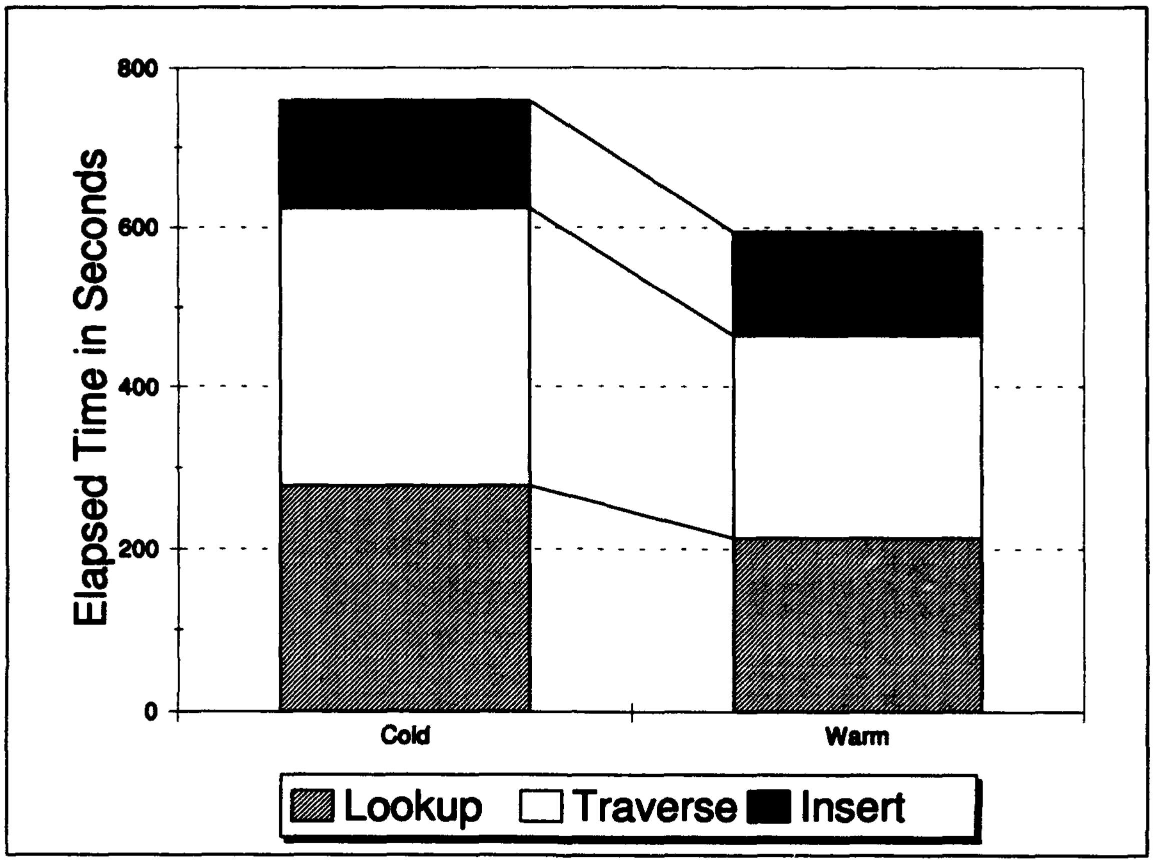

Figure 30. Itasca OO1 Average Benchmark Results for *Small Remote* Database

Table 20. Itasca OO1 Normalized Reverse Traversal Results for *Small Remote* Database

Benchmark	Cold	Warm
1	250.596	221.466
2	4149.486	330.472
3	5182.592	313.930
4	267.115	825.788
5	650.879	1528.182
Average:	2100.134	643.968
Sample STD:	2376.051	547.861

Table 21. Itasca OO1 Raw Benchmark Results for *Small Remote* Database

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	236.436	205.299	194.476	200.923	191.730	203.783	192.261	171.489	197.298	184.745
	Traversal	259.067	242.228	216.204	204.268	199.765	208.713	210.271	209.419	213.249	217.469
	Insert	135.861	121.657	118.344	158.288	125.882	129.101	131.486	135.400	138.727	132.461
	R. Traversal	647.577	408.563	113.341	80.429	333.873	115.308	561.862	375.284	138.720	229.127
2	Lookup	237.550	203.369	189.486	200.923	228.893	197.294	191.996	172.255	192.125	182.931
	Traversal	311.968	214.625	239.053	210.905	203.646	245.218	207.455	212.828	238.216	200.333
	Insert	135.286	189.544	118.256	122.962	126.706	124.665	122.928	127.653	130.864	127.095
	R. Traversal	3.795	279.790	0.325	321.790	317.528	111.475	413.313	320.441	356.210	116.004
3	Lookup	241.633	211.237	197.420	204.214	303.894	191.700	202.497	188.985	194.717	273.129
	Traversal	265.949	218.323	345.212	234.268	209.654	311.801	243.373	215.695	322.601	214.836
	Insert	131.300	121.421	119.250	122.527	125.272	130.232	128.198	133.618	136.939	138.019
	R. Traversal	3.160	361.918	524.406	250.292	269.360	186.745	538.020	306.622	68.926	202.247
4	Lookup	248.285	204.900	191.615	198.560	373.604	206.994	194.545	174.698	193.305	185.239
	Traversal	404.054	214.123	212.854	394.783	234.031	335.992	380.636	224.529	218.874	391.595
	Insert	136.238	127.289	136.653	122.229	131.554	126.413	125.009	135.835	135.090	105.371
	R. Traversal	300.912	110.249	145.009	394.779	260.322	150.371	136.799	594.040	3.137	400.945
5	Lookup	422.488	393.502	214.008	208.158	199.106	177.196	412.045	202.776	201.272	191.693
	Traversal	494.163	242.194	227.206	223.970	472.039	223.705	212.763	438.037	231.114	213.075
	Insert	135.139	119.709	119.042	120.133	125.240	128.894	130.805	135.203	141.677	143.425
	R. Traversal	370.882	289.201	176.920	527.393	3.382	171.806	204.635	195.620	354.774	216.239

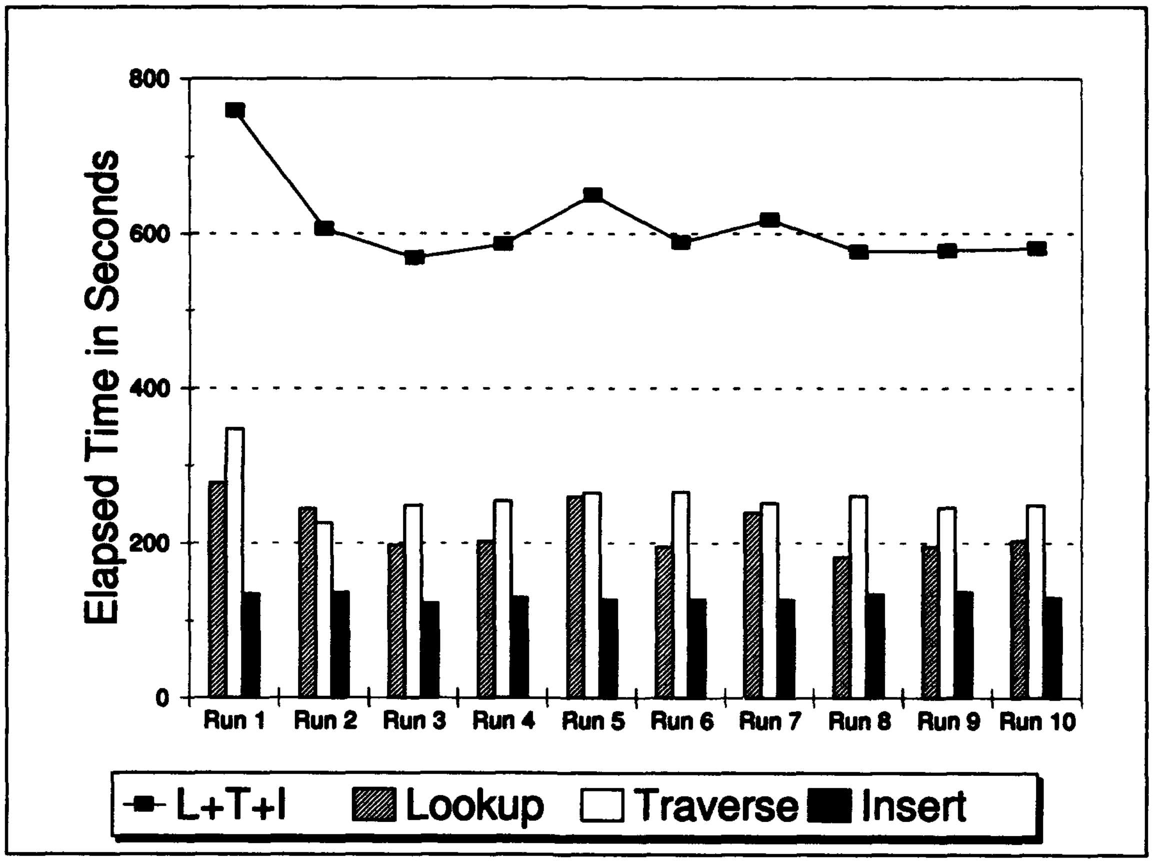

Figure 31. Itasca OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Remote* Database

Table 22. Itasca OO1 Summary Results for *Small Local* Database

	Lookup		Traversal		Insert		L+T+I	
Benchmark	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	240.378	193.092	260.325	196.309	135.420	118.780	636.123	508.181
2	228.906	199.370	242.878	219.310	128.010	121.717	599.794	540.397
3	242.608	219.948	246.443	241.543	134.228	121.310	623.279	582.801
4	241.792	194.821	245.327	193.768	129.075	119.659	616.194	508.248
5	301.635	193.305	249.863	207.916	127.542	120.684	679.040	521.905
Average:	251.064	200.107	248.967	211.769	130.855	120.430	630.886	532.306
Sample STD:	28.810	11.376	6.828	19.502	3.690	1.205	29.944	31.162

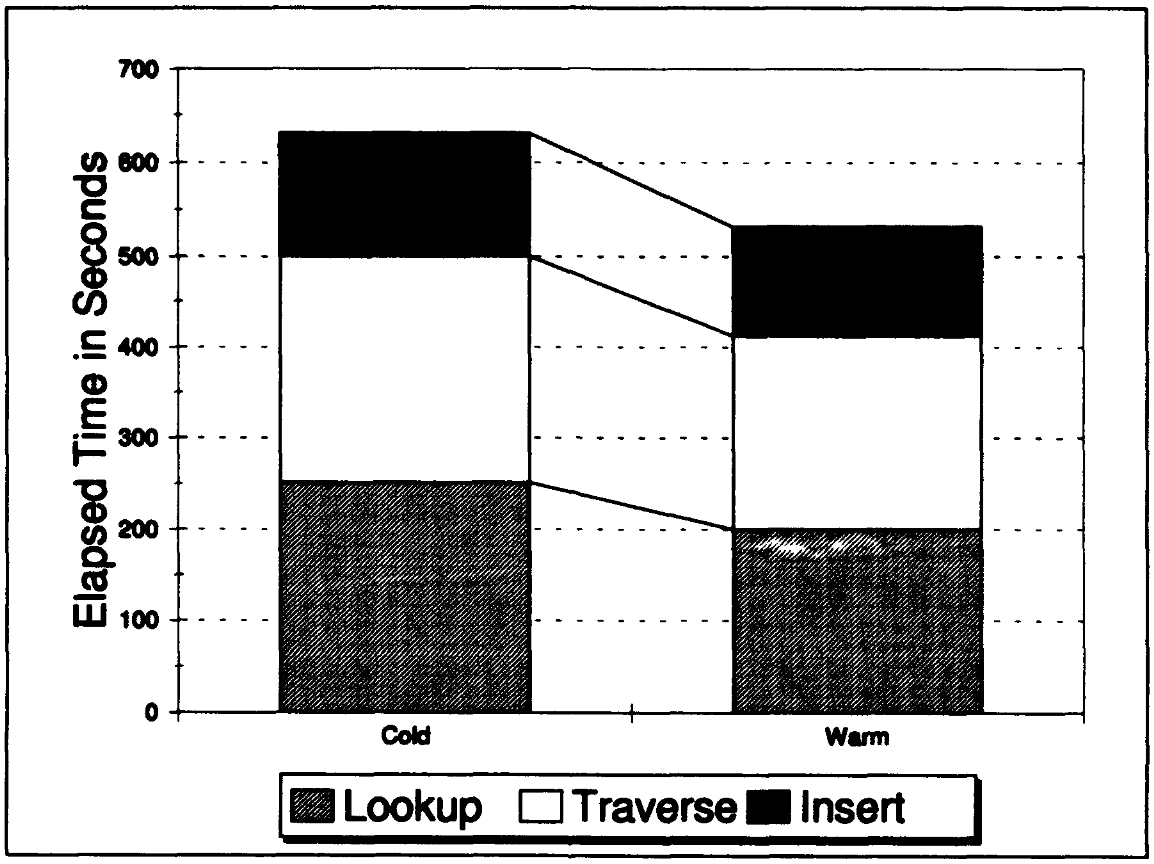

Figure 32. Itasca OO1 Average Benchmark Results for *Small Local* Database

Table 23. Itasca OO1 Normalized Reverse Traversal Results for *Small Local* Database

Benchmark	Cold	Warm
1	227.666	211.203
2	4119.775	1071.200
3	5817.336	310.269
4	258.015	776.981
5	381.341	317.110
Average:	2160.827	537.353
Sample STD:	2633.055	370.426

Table 24. Itasca OO1 Raw Benchmark Results for *Small Local* Database

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	240.378	201.908	212.243	197.746	188.187	195.231	198.186	168.695	192.523	183.114
	Traversal	260.325	197.758	199.725	178.984	194.230	189.535	194.863	199.358	186.890	225.435
	Insert	135.420	114.477	113.987	112.180	111.746	120.956	121.984	125.572	127.599	120.516
	R. Traversal	588.323	388.668	109.774	61.703	365.246	113.260	533.887	336.550	124.876	258.912
2	Lookup	228.906	203.339	242.536	199.541	190.333	196.724	192.329	190.654	196.493	182.384
	Traversal	242.878	272.312	208.313	191.279	258.254	217.725	187.836	271.174	188.801	178.097
	Insert	128.010	116.235	112.429	115.347	118.118	123.792	125.385	127.335	131.033	125.776
	R. Traversal	3.768	367.148	2.329	226.444	268.600	195.237	324.930	274.017	424.523	123.875
3	Lookup	242.608	206.322	323.230	219.087	195.355	175.699	192.939	287.853	194.006	185.042
	Traversal	246.443	342.283	203.751	191.158	323.821	197.253	197.460	334.735	207.106	176.322
	Insert	134.228	117.598	111.380	113.370	112.161	122.145	126.095	127.265	130.263	131.510
	R. Traversal	3.547	470.443	334.603	394.087	128.000	148.253	500.567	264.635	226.572	91.392
4	Lookup	241.792	204.451	191.600	199.442	192.512	211.966	190.032	173.585	197.886	191.914
	Traversal	245.327	185.112	192.288	193.309	188.458	212.052	187.346	197.875	188.475	198.992
	Insert	129.075	113.674	115.886	118.753	122.078	123.038	126.433	131.924	128.373	96.770
	R. Traversal	290.660	98.427	132.533	168.697	240.357	116.503	105.735	342.915	3.271	290.755
5	Lookup	301.635	216.181	188.778	197.445	193.860	172.769	194.529	194.826	193.651	187.703
	Traversal	249.863	229.010	191.945	194.084	232.393	196.017	192.818	243.741	201.009	190.228
	Insert	127.542	113.733	111.199	114.401	115.330	115.518	124.848	127.973	130.736	132.415
	R. Traversal	217.295	256.820	162.961	220.844	0.337	201.762	187.949	181.021	82.329	230.502

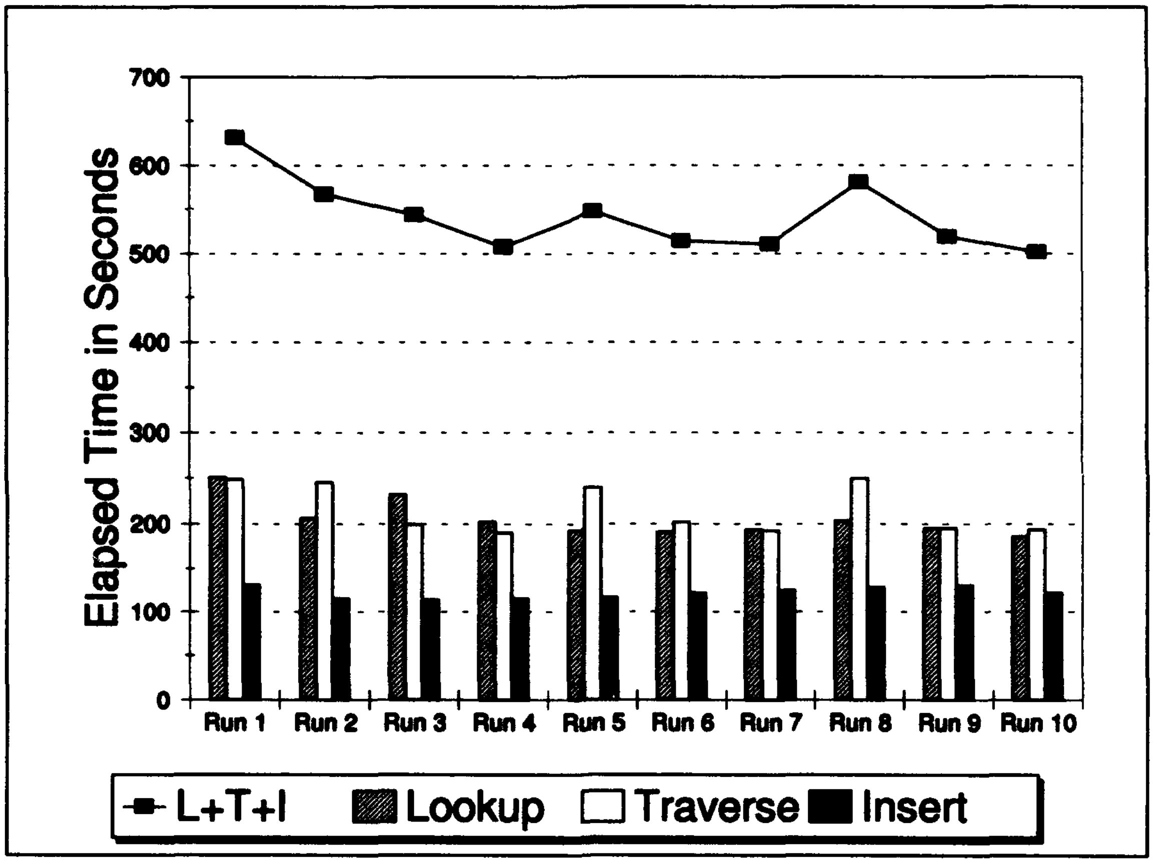

Figure 33. **Itasca OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database**

Table 25. Itasca OO1 Summary Results for *Small Remote* Database (NLOR)

Benchmark	Lookup		Traversal		Insert		L+T+I	
	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	250.159	196.670	339.031	249.512	181.419	156.379	770.609	602.561
2	293.851	194.376	376.279	325.467	187.357	171.738	857.487	691.581
3	246.583	229.023	466.312	367.915	188.729	157.161	901.624	754.099
4	225.023	199.118	319.697	253.406	188.773	161.897	733.493	614.421
5	246.787	200.589	399.764	309.017	283.734	157.592	930.285	667.198
Average:	252.481	203.955	380.217	301.063	206.002	160.953	838.700	665.972
Sample STD:	25.189	14.212	57.378	50.142	43.559	6.399	84.269	61.445

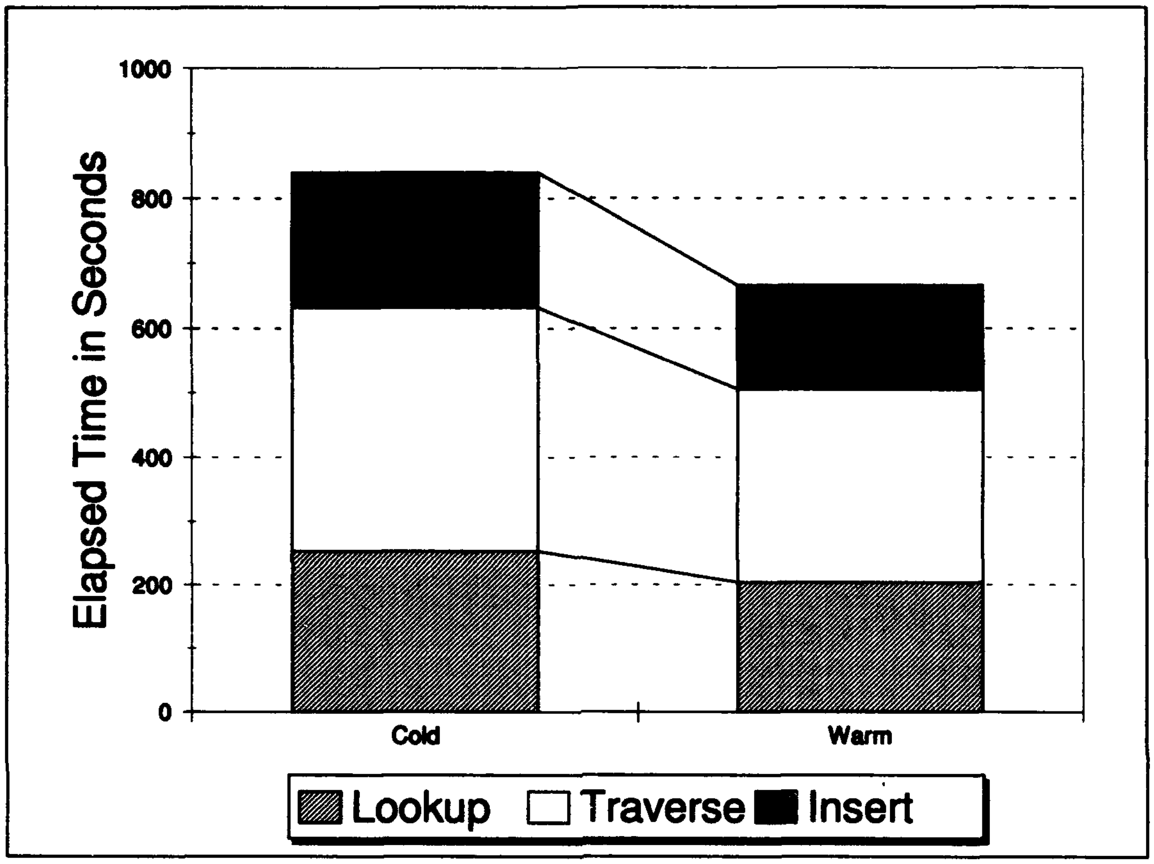

Figure 34. Itasca OO1 Average Benchmark Results for *Small Remote* Database (NLOR)

Table 26. Itasca OO1 Normalized Reverse Traversal Results for *Small Remote* Database (NLOR)

Benchmark	Cold	Warm
1	303.841	424.245
2	438.734	1573.076
3	530.117	400.771
4	332.671	271.608
5	401.202	309.705
Average:	401.313	595.881
Sample STD:	89.726	549.883

Table 27. Itasca OO1 Raw Benchmark Results for *Small Remote* Database (NLOR)

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	250.159	209.790	198.968	202.836	196.738	183.067	196.630	186.442	200.357	195.207
	Traversal	339.031	261.028	251.398	250.129	239.966	262.830	251.099	242.961	239.704	246.492
	Insert	181.419	165.603	159.465	158.267	148.363	153.240	157.681	151.869	157.337	155.587
	R. Traversal	990.724	318.601	465.033	583.243	261.968	235.338	169.638	375.228	0.447	203.994
2	Lookup	293.851	208.546	185.879	203.539	181.687	214.983	182.901	196.902	180.122	194.820
	Traversal	376.279	352.569	259.570	353.656	347.392	359.838	287.138	355.360	255.518	358.160
	Insert	187.357	289.457	173.416	163.057	151.002	151.797	153.887	147.563	159.050	156.409
	R. Traversal	147.672	840.254	3.492	160.036	794.727	605.615	320.830	352.589	345.140	152.153
3	Lookup	246.583	208.545	296.303	238.664	193.547	197.035	185.678	331.267	209.743	200.426
	Traversal	466.312	267.190	398.486	398.070	282.412	422.240	415.233	267.343	425.301	434.961
	Insert	188.729	169.750	158.606	155.517	152.637	156.219	157.477	150.408	157.979	155.857
	R. Traversal	120.731	165.316	357.452	782.827	453.292	231.260	477.293	200.141	373.684	61.743
4	Lookup	225.023	211.491	189.405	203.405	188.423	210.226	187.260	201.223	197.072	203.554
	Traversal	319.697	256.122	257.862	248.145	269.898	271.802	247.013	236.321	244.749	248.740
	Insert	188.773	170.346	162.187	190.502	152.855	159.566	156.400	149.471	157.426	158.317
	R. Traversal	316.038	388.546	285.191	140.435	440.151	294.172	215.487	354.863	88.149	525.867
5	Lookup	246.787	210.180	238.942	199.577	193.194	198.959	190.902	191.437	198.614	183.497
	Traversal	399.764	256.127	334.464	326.168	251.027	338.463	328.941	256.662	347.362	341.941
	Insert	283.734	169.298	161.945	156.922	154.336	156.117	157.434	149.451	156.828	156.000
	R. Traversal	509.698	187.001	46.164	760.234	387.349	99.430	1100.549	586.670	577.801	214.548

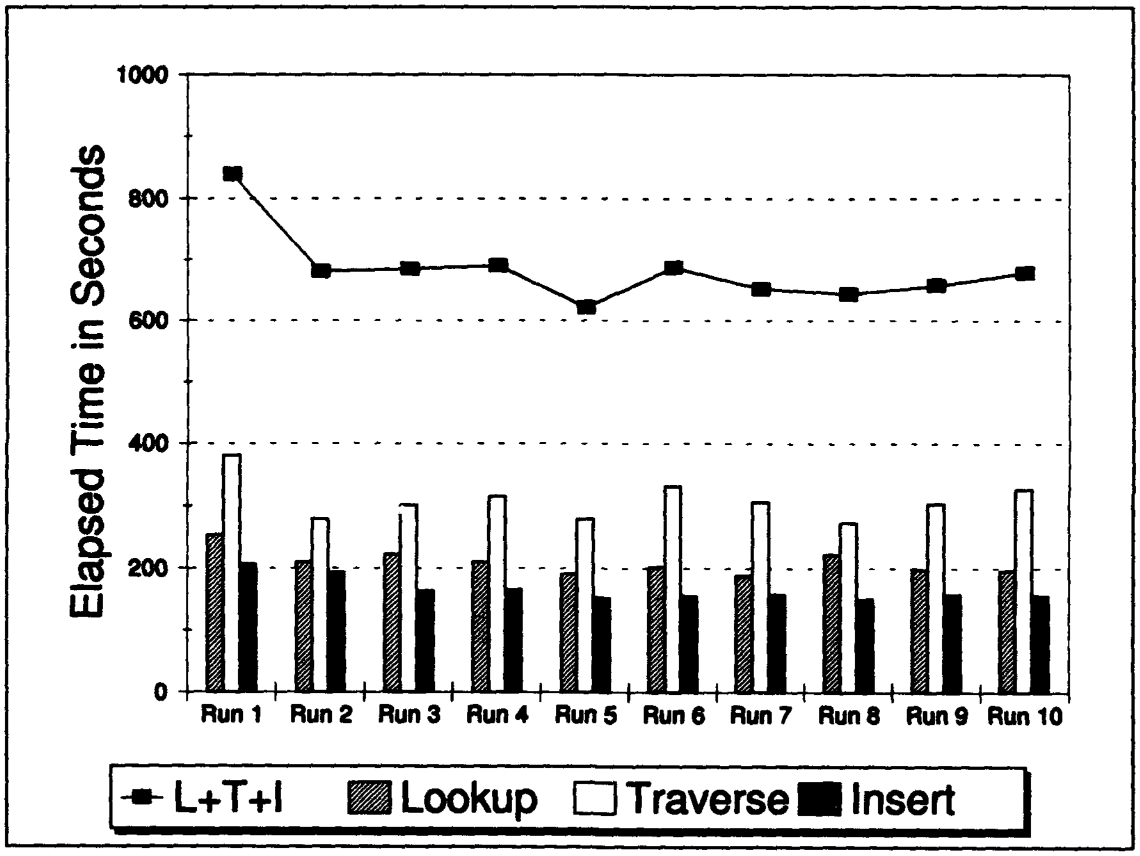

Figure 35. Itasca OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Remote* Database (NLOR)

Table 28. Itasca OO1 Summary Results for *Small Local* Database (NLOR)

Benchmark	Lookup		Traversal		Insert		L+T+I	
	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	234.236	197.707	342.590	270.023	188.591	150.635	765.417	618.365
Average:	234.236	197.707	342.590	270.023	188.591	150.635	765.417	618.365
Sample STD:	*none*	*none*	*none*	*none*	*none*	*none*	*none*	*none*

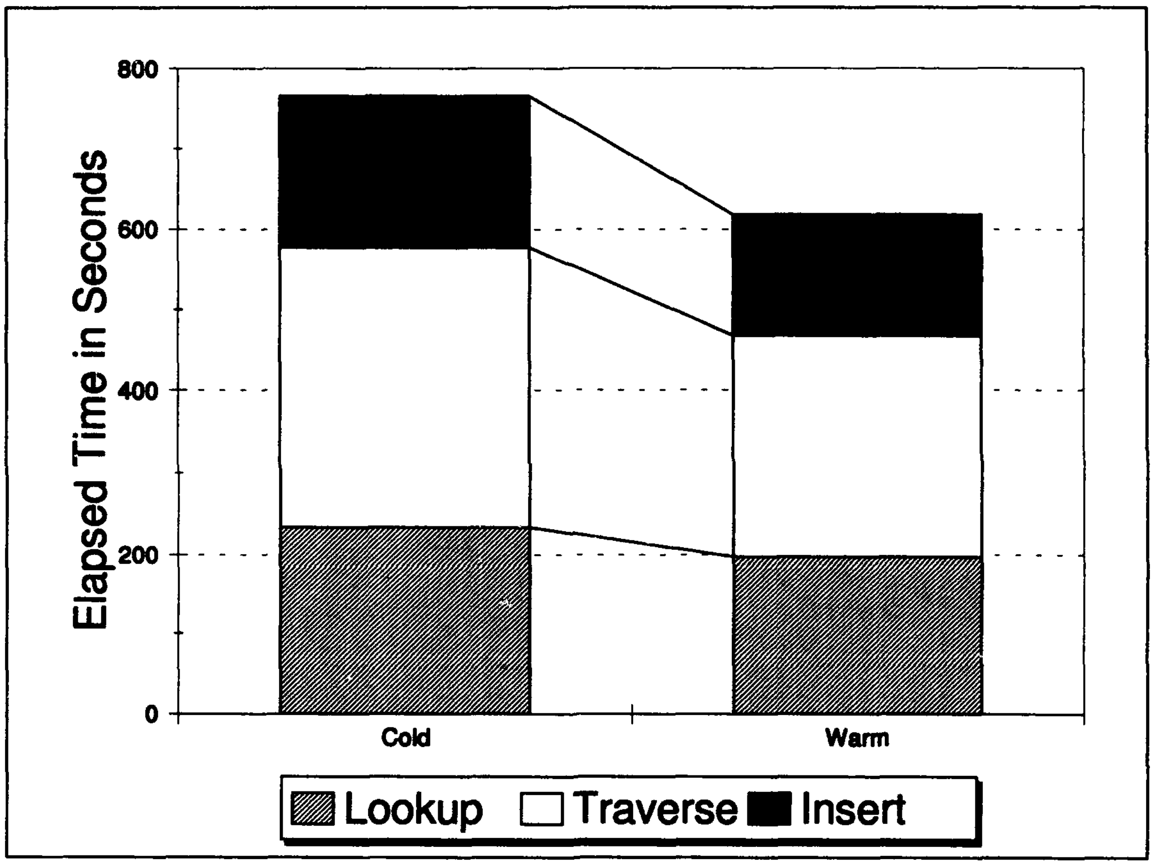

Figure 36. Itasca OO1 Average Benchmark Results for *Small Local* Database (NLOR)

Table 29. Itasca OO1 Raw Benchmark Results for *Small Local* Database (NLOR)

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	234.236	236.356	196.240	201.056	195.222	177.328	206.415	183.878	195.434	187.437
	Traversal	342.590	265.518	257.364	231.045	262.865	289.085	234.690	289.329	297.341	302.968
	Insert	188.591	160.826	155.843	152.629	139.897	144.978	153.243	145.895	151.140	151.262
2	Lookup	351.032	212.306	182.788	199.951	180.810	310.290	188.885	196.601	178.093	282.823
	Insert	189.797	164.439	154.720	152.014	143.154	148.015	152.120	141.279	151.884	151.955
3	Lookup	401.437	210.438	186.554	200.711	337.687	214.021	185.262	198.132	180.787	374.024
	Insert	192.812	158.573	156.594	149.858	141.376	145.605	375.009	172.273	170.564	156.810

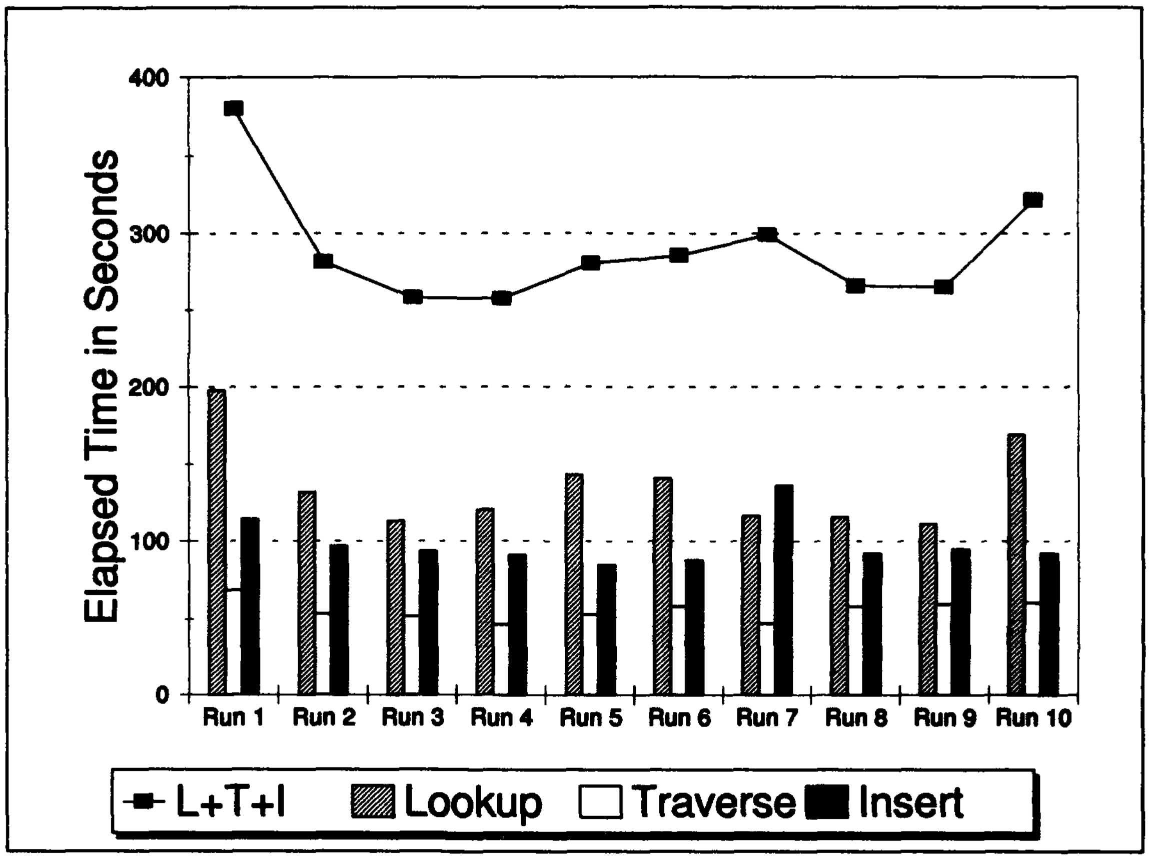

Figure 37. Itasca OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database (NLOR)

B.2 OO1 Benchmark Results for the Matisse DBMS

This section reports our OO1 benchmark results for the Matisse DBMS. Some benchmark runs were duplicated to determine the impact of the *mts_collect_versions* program, required by Matisse to compact disk space in the database, on performance. Results for the following database configurations are provided:

- Matisse *Small Remote* Database — The results for this benchmark configuration are reported in Tables 30, 31, and 32 and in Figures 38 and 39. The *mts_collect_versions* program was executed after each benchmark iteration with no delay between the execution and the start of the next benchmark iteration.

- Matisse *Small Local* Database — The results for this benchmark configuration are reported in Tables 33, 34, and 35 and in Figures 40 and 41. The *mts_collect_versions* program was executed after each benchmark iteration with no delay between the execution and the start of the next benchmark iteration.

- Matisse *Small Local* Database (200 second delay) — The results for this benchmark configuration are reported in Tables 36, 37, and 38 and in Figures 42 and 43. The *mts_collect_versions* program was executed after each benchmark iteration. A 200 second delay from the start of the version collection program to the start of the next benchmark iteration was done to allow version collection without iterference from the benchmark program.

- Matisse *Small Remote* Database with No Locality of Reference (NLOR) — The results for this benchmark configuration are reported in Tables 39, 40, and 41 and in Figures 44 and 45. The *mts_collect_versions* program was executed after each benchmark iteration with no delay between the execution and the start of the next benchmark iteration.

- Matisse *Small Remote* Database with No Locality of Reference (200 second delay)— The results for this benchmark configuration are reported in Tables 42, 43, and 44 and in Figures 46 and 47. The *mts_collect_versions* program was executed after each benchmark iteration. A 200 second delay from the start of the version collection program to the start of the next benchmark iteration was done to allow version collection without iterference from the benchmark program.

- Matisse *Small Local* Database with No Locality of Reference — The results for this benchmark configuration are reported in Tables 45, 46, and 47 and in Figures 48 and 49. The *mts_collect_versions* program was executed after each benchmark iteration with no delay between the execution and the start of the next benchmark iteration.
- Matisse *Large Remote* Database — The results for this benchmark configuration are reported in Tables 48, 49, and 50 and in Figures 50 and 51. Only four out of five of the benchmark runs completed in this configuration due to problems with the Matisse server.
- Matisse *Large Local* Database — The results for this benchmark configuration are reported in Tables 51, 52, and 53 and in Figures 52 and 53. Only four out of five of the benchmark runs completed in this configuration due to problems with the Matisse server.

Table 30. Matisse OO1 Summary Results for *Small Remote* Database

Benchmark	Lookup		Traversal		Insert		L+T+I	
	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	64.540	38.694	58.105	36.079	61.337	50.173	183.982	124.946
2	148.535	75.705	56.612	34.543	64.900	50.665	270.047	160.913
3	132.268	75.343	54.405	35.584	65.707	48.111	252.380	159.038
4	147.820	74.349	60.233	34.453	66.867	48.329	274.920	157.131
5	136.746	78.349	36.202	36.278	61.568	50.256	234.516	164.883
Average:	125.982	68.448	53.111	35.387	64.076	49.507	243.169	153.382
Sample STD:	35.057	16.721	9.689	0.851	2.496	1.192	36.720	16.153

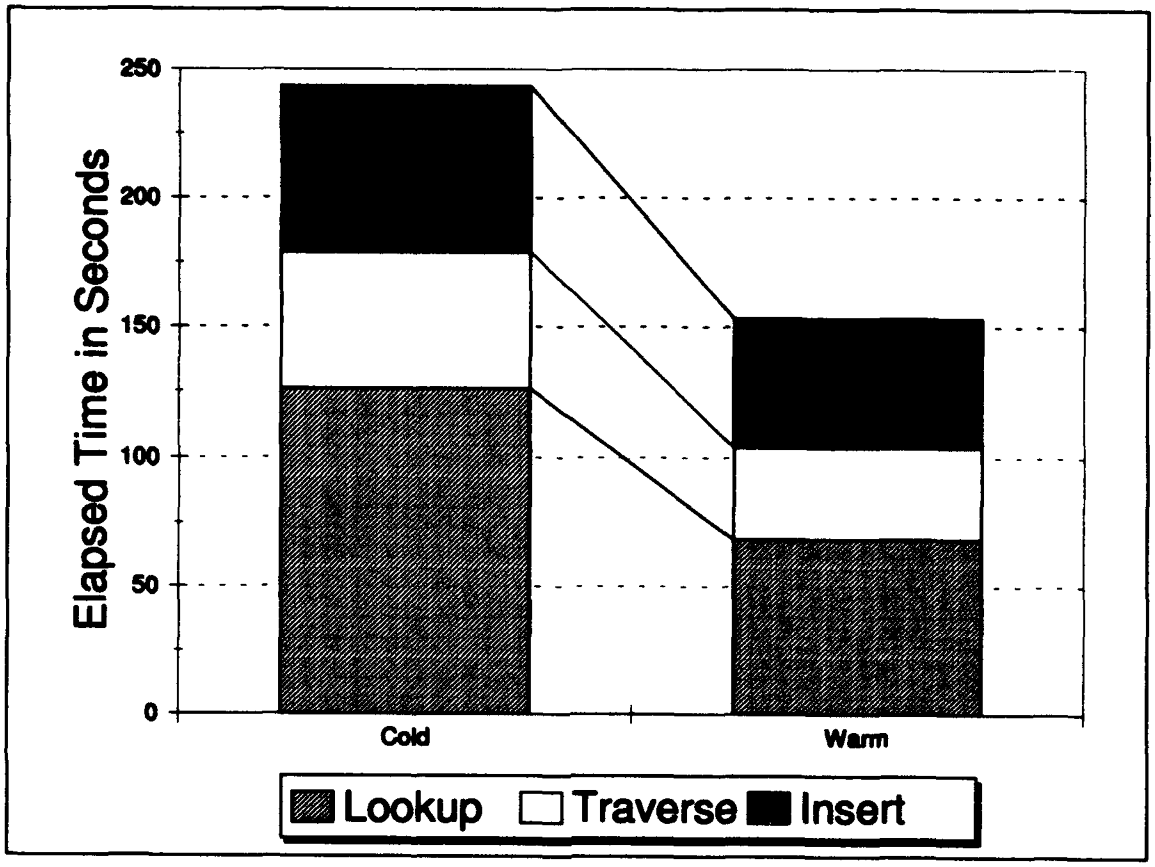

Figure 38. Matisse OO1 Average Benchmark Results for *Small Remote* Database

Table 31. Matisse OO1 Normalized Reverse Traversal Results for *Small Remote* Database

Benchmark	Cold	Warm
1	44.402	40.358
2	65.864	44.055
3	70.291	88.096
4	67.583	77.150
5	56.491	39.075
Average:	60.926	57.747
Sample STD:	10.596	23.109

Table 32. Matisse OO1 Raw Benchmark Results for *Small Remote* Database

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	64.540	50.932	47.430	46.041	42.034	32.934	33.221	31.687	31.850	32.118
	Traversal	58.105	41.301	34.556	38.288	25.844	38.031	38.790	35.268	38.595	34.042
	Insert	61.337	59.722	54.843	53.046	47.873	49.073	48.703	45.915	45.926	46.453
	R. Traversal	82.644	45.561	53.788	17.759	43.587	27.474	47.989	23.688	43.547	9.764
2	Lookup	148.535	94.954	97.046	103.014	110.361	104.688	74.393	32.443	32.148	32.295
	Traversal	56.612	37.822	32.045	38.620	37.253	28.049	33.665	34.875	26.383	42.178
	Insert	64.900	58.705	54.376	52.265	56.261	48.638	44.894	46.595	47.053	47.198
	R. Traversal	35.804	98.282	11.216	56.830	66.519	51.653	61.896	53.120	29.053	20.865
3	Lookup	132.268	91.154	101.178	93.530	108.769	103.710	83.656	32.037	32.072	31.983
	Traversal	54.405	46.104	40.450	35.847	30.505	37.366	34.425	34.547	34.651	26.364
	Insert	65.707	54.409	51.760	49.587	46.361	46.144	45.949	45.991	45.827	46.975
	R. Traversal	30.988	127.010	50.249	12.141	0.141	91.076	40.685	60.505	61.869	6.830
4	Lookup	147.820	103.912	98.034	106.216	93.359	82.746	88.121	31.797	32.600	32.354
	Traversal	60.233	43.045	38.355	36.618	29.177	44.109	26.926	26.585	33.158	32.103
	Insert	66.867	54.512	50.940	49.833	47.292	47.286	45.707	46.880	48.025	44.483
	R. Traversal	43.785	71.732	0.339	52.574	3.096	34.209	50.166	57.608	14.695	19.090
5	Lookup	136.746	143.337	113.650	108.523	106.469	105.112	32.223	31.962	32.009	31.853
	Traversal	36.202	42.623	45.941	41.741	35.483	29.553	32.773	32.209	33.180	33.001
	Insert	61.568	56.559	52.132	49.002	47.708	47.779	53.009	46.312	53.728	46.070
	R. Traversal	52.753	53.457	49.247	47.440	51.657	21.422	5.741	40.156	15.562	37.742

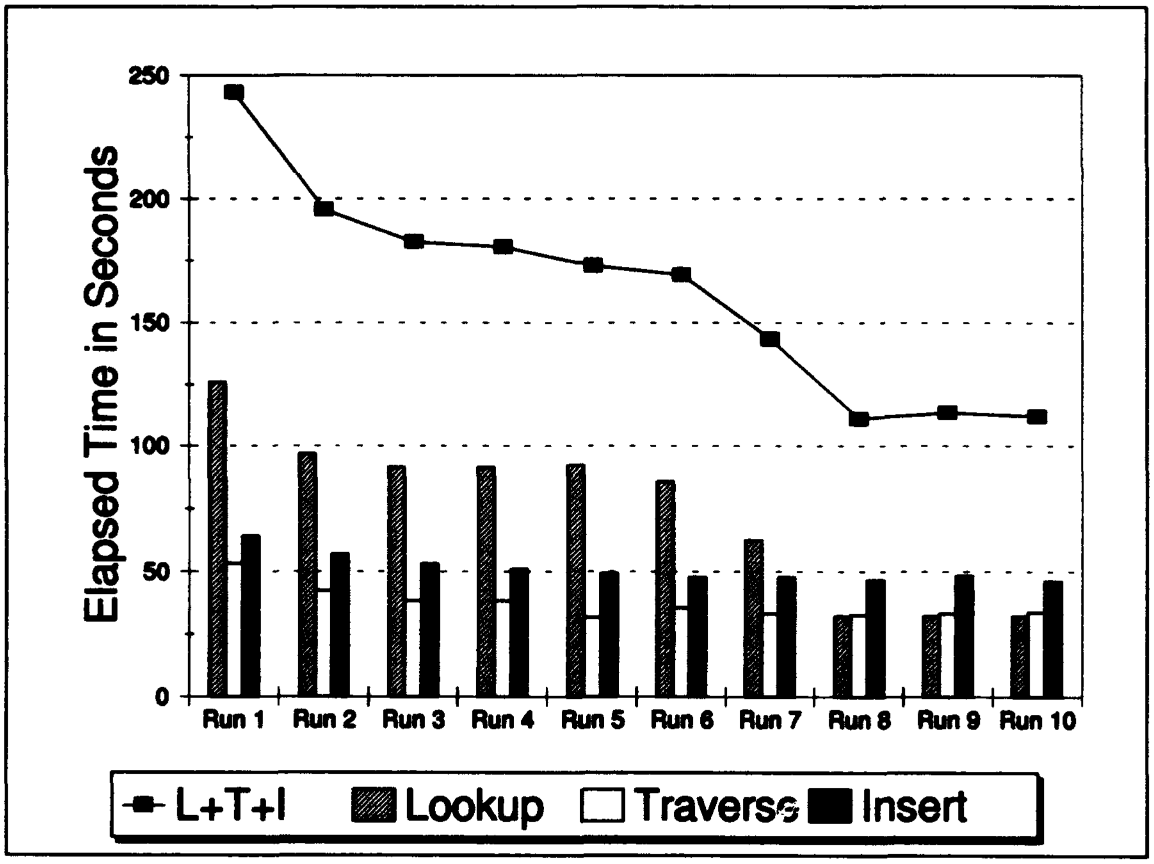

Figure 39. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Remote* Database

Table 33. Matisse OO1 Summary Results for *Small Local* Database

	Lookup		Traversal		Insert		L+T+I	
Benchmark	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	59.272	32.796	55.632	30.445	61.076	47.353	175.980	110.594
2	102.436	44.126	77.500	38.633	64.505	50.674	244.441	133.433
3	91.476	29.807	83.428	45.500	63.403	49.324	238.307	124.631
4	99.040	30.191	85.638	47.903	66.363	51.308	251.041	129.402
5	106.038	43.372	50.299	42.667	66.048	50.837	222.385	136.876
Average:	91.652	36.058	70.499	41.030	64.279	49.899	226.431	126.987
Sample STD:	18.883	7.119	16.389	6.850	2.153	1.603	30.136	10.240

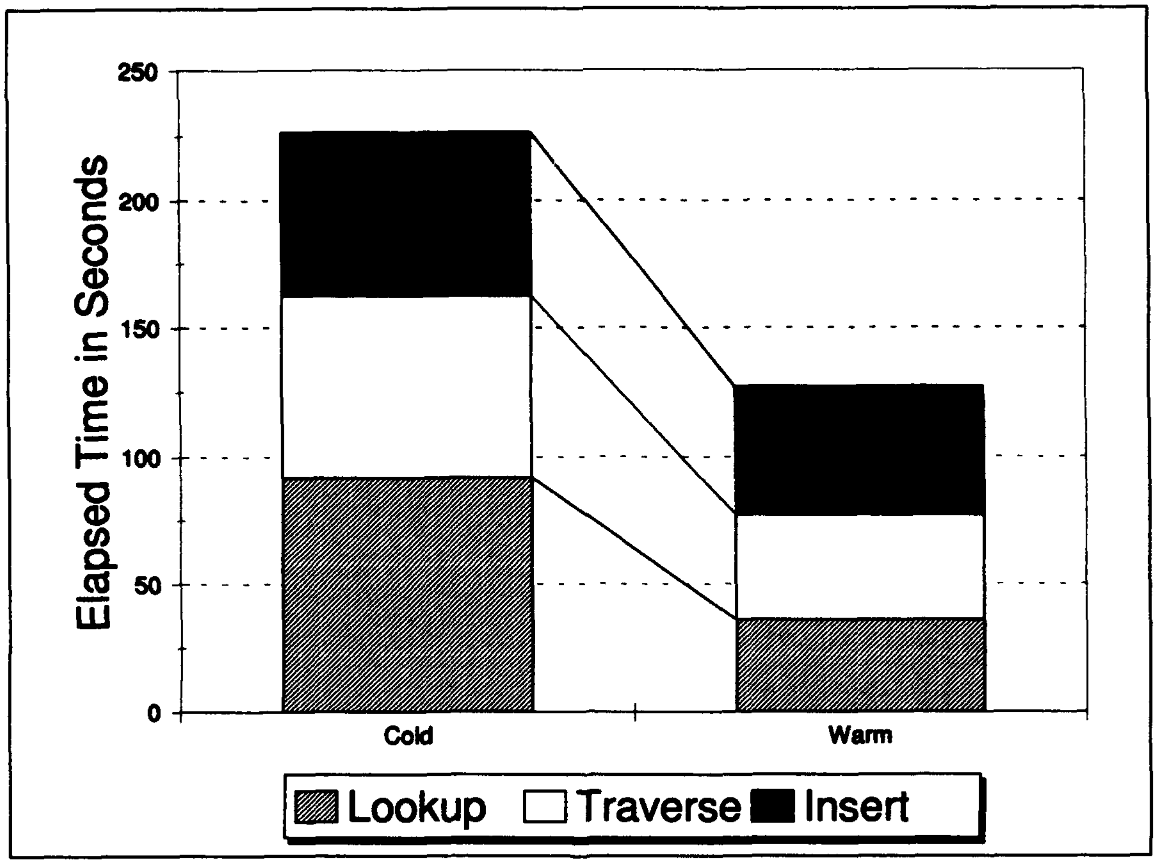

Figure 40. Matisse OO1 Average Benchmark Results for *Small Local* Database

Table 34. Matisse OO1 Normalized Reverse Traversal Results for *Small Local* Database

Benchmark	Cold	Warm
1	48.944	36.430
2	68.145	33.792
3	89.006	82.310
4	84.334	67.425
5	57.537	32.891
Average:	69.593	50.570
Sample STD:	17.088	22.834

Table 35. Matisse OO1 Raw Benchmark Results for *Small Local* Database

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	59.272	44.054	37.780	35.598	34.985	31.270	28.310	27.677	27.715	27.774
	Traversal	55.632	38.808	30.431	31.631	22.388	30.818	31.733	29.158	31.184	27.854
	Insert	61.076	53.396	49.652	47.667	45.686	46.000	45.776	46.381	47.246	44.372
	R. Traversal	91.098	41.795	51.207	17.680	43.076	25.318	40.951	19.810	35.758	8.144
2	Lookup	102.436	76.352	81.278	68.078	29.771	28.275	28.399	28.297	28.271	28.414
	Traversal	77.500	47.718	43.710	46.175	35.991	31.163	36.240	35.755	33.513	37.434
	Insert	64.505	57.408	53.377	52.013	48.457	49.653	50.617	48.224	47.750	48.570
	R. Traversal	37.043	73.722	7.599	38.891	51.596	41.536	51.168	44.124	24.105	17.035
3	Lookup	91.476	35.784	29.490	29.097	29.569	28.597	29.594	28.921	28.162	29.051
	Traversal	83.428	70.456	58.785	45.981	42.961	42.242	39.224	37.366	40.096	32.392
	Insert	63.403	57.983	52.451	51.885	49.271	47.941	46.979	45.712	45.855	45.844
	R. Traversal	39.238	152.596	39.557	9.259	0.141	73.227	32.634	51.650	50.748	5.432
4	Lookup	99.040	42.881	30.019	28.818	28.642	28.220	27.865	27.775	29.293	28.207
	Traversal	85.638	66.943	65.926	55.021	40.001	54.410	37.276	33.621	42.922	35.011
	Insert	66.363	58.538	59.802	50.938	50.163	50.654	49.216	47.170	48.080	47.208
	R. Traversal	54.637	60.911	0.309	45.530	2.465	29.545	47.398	52.970	12.730	15.319
5	Lookup	106.038	99.278	87.224	33.146	28.741	28.491	28.492	28.375	28.445	28.153
	Traversal	50.299	58.054	47.754	47.719	46.195	33.779	34.514	37.916	40.272	37.797
	Insert	66.048	59.695	53.629	51.861	49.777	48.993	47.375	47.624	51.436	47.145
	R. Traversal	53.730	49.077	42.124	40.055	42.801	18.091	4.708	32.838	13.050	31.106

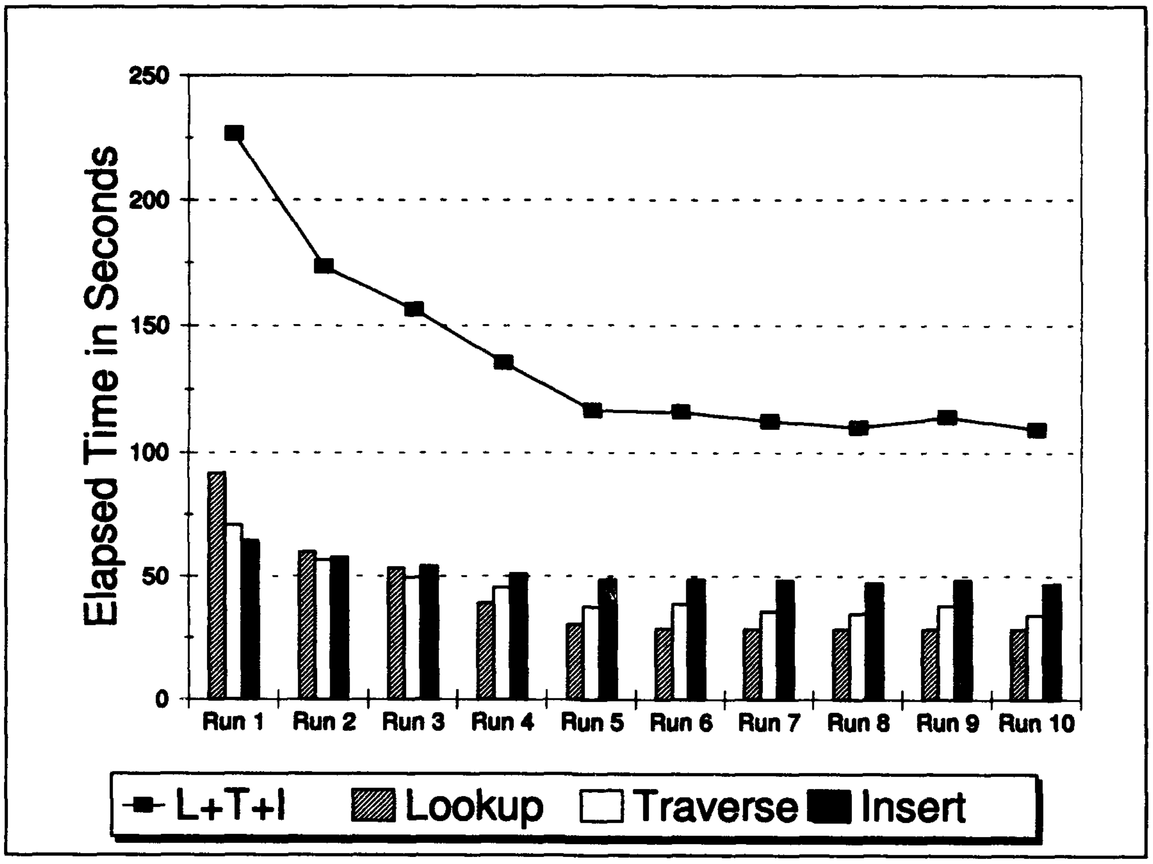

Figure 41. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database

Table 36. Matisse OO1 Summary Results for *Small Local* Database 200 Second Delay for Version Collection

	Lookup		Traversal		Insert		L+T+I	
Benchmark	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	165.410	34.576	118.109	49.526	87.240	54.480	370.759	138.582
2	63.257	30.640	52.195	28.188	60.113	44.940	175.565	103.768
3	118.390	30.874	74.919	48.950	60.436	47.032	253.745	126.856
4	128.212	35.505	75.568	31.923	61.852	47.572	265.632	115.000
5	89.581	29.302	56.326	35.052	62.839	48.066	208.746	112.420
Average:	112.970	32.179	75.423	38.728	66.496	48.418	254.889	119.325
Sample STD:	38.827	2.700	26.108	9.899	11.648	3.592	74.101	13.564

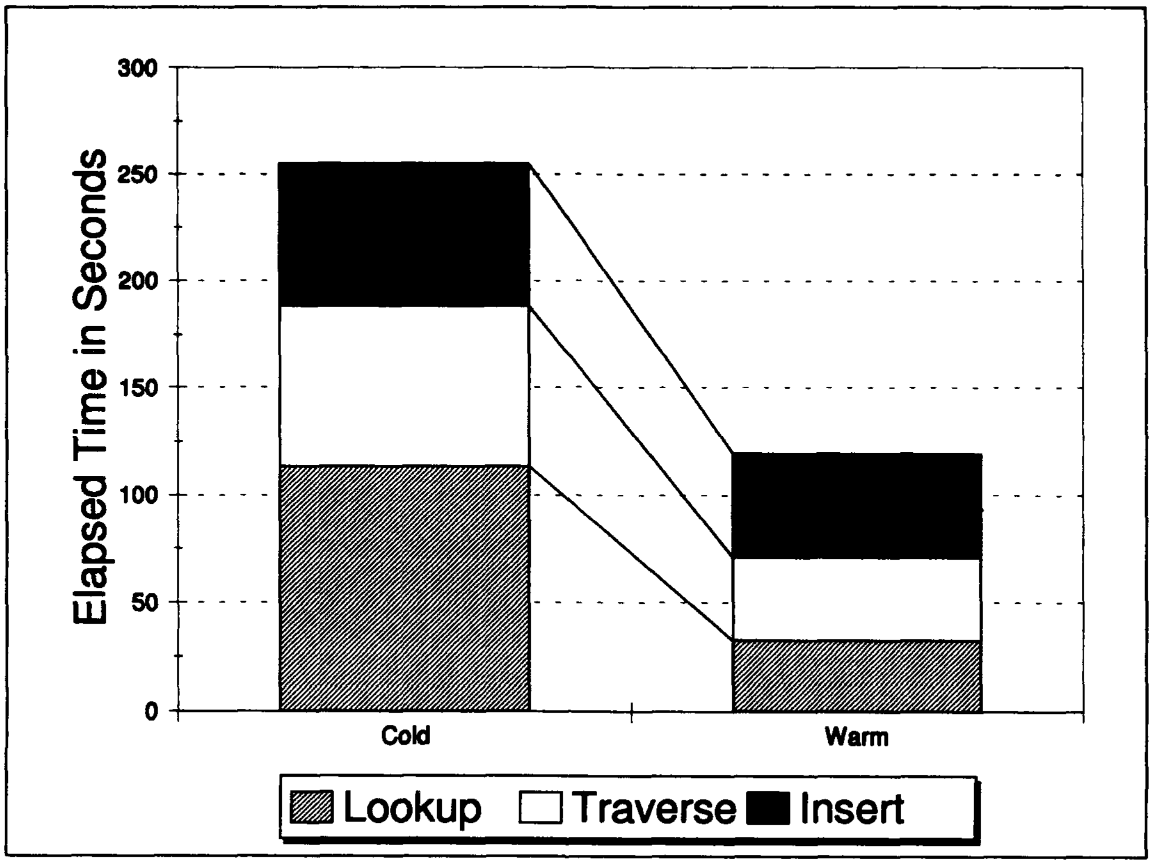

Figure 42. Matisse OO1 Average Benchmark Results for *Small Local* Database 200 Second Delay for Version Collection

Table 37. Matisse OO1 Normalized Reverse Traversal Results for *Small Local* Database 200 Second Delay for Version Collection

Benchmark	Cold	Warm
1	101.904	40.405
2	67.926	38.124
3	113.111	100.268
4	65.571	56.479
5	56.599	32.325
Average:	81.022	53.520
Sample STD:	24.862	27.624

Table 38. Matisse OO1 Raw Benchmark Results for *Small Local* Database 200 Second Delay for Version Collection

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	165.410	83.504	30.344	28.607	27.868	27.950	28.490	28.056	28.216	28.149
	Traversal	118.109	72.372	69.301	55.145	35.031	49.669	44.496	34.976	40.979	43.765
	Insert	87.240	79.981	54.287	52.789	51.372	51.183	49.946	50.023	52.394	48.342
	R. Traversal	189.673	64.902	52.699	17.536	44.348	23.719	43.141	22.484	38.033	9.782
2	Lookup	63.257	45.180	35.784	28.379	27.908	27.661	28.086	27.435	27.690	27.638
	Traversal	52.195	30.858	26.441	31.118	30.370	22.828	27.326	28.126	22.646	33.982
	Insert	60.113	50.784	46.737	45.211	45.216	44.427	42.275	42.583	42.290	44.935
	R. Traversal	36.924	87.346	9.557	49.473	65.958	45.501	52.134	43.683	23.870	17.051
3	Lookup	118.390	49.455	31.143	28.924	28.646	28.046	28.523	27.981	28.014	27.134
	Traversal	74.919	68.675	66.241	55.473	44.711	46.070	39.971	41.294	41.403	36.715
	Insert	60.436	54.829	49.720	48.288	45.774	43.662	44.781	45.637	46.224	44.372
	R. Traversal	49.866	207.563	69.812	13.917	0.160	80.529	36.319	77.281	62.322	7.704
4	Lookup	128.212	80.148	33.600	30.760	29.484	30.035	28.423	29.103	28.604	29.392
	Traversal	75.568	41.011	37.057	36.133	28.781	42.104	24.108	23.485	27.885	26.741
	Insert	61.852	52.746	52.137	46.071	45.445	46.593	49.648	45.443	45.518	44.545
	R. Traversal	42.481	57.647	0.253	38.166	1.731	23.909	39.532	48.599	11.952	14.951
5	Lookup	89.581	35.458	30.867	29.671	27.856	27.655	27.658	28.419	27.783	28.351
	Traversal	56.326	67.421	57.205	32.705	27.213	24.019	26.787	26.454	27.002	26.658
	Insert	62.839	55.566	52.003	49.706	49.037	47.197	45.674	43.640	45.444	44.323
	R. Traversal	52.854	46.031	41.469	39.118	41.942	17.636	4.809	32.498	12.941	30.588

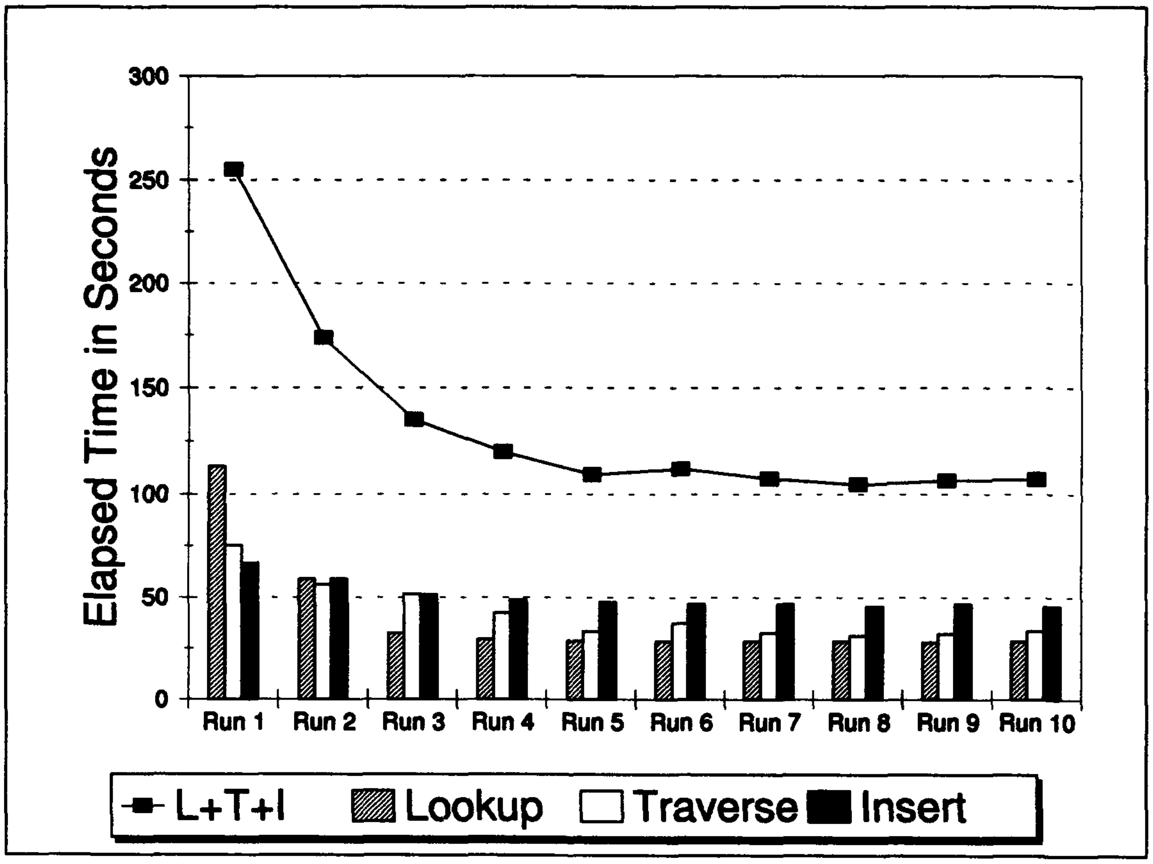

Figure 43. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database 200 Second Delay for Version Collection

Table 39. Matisse OO1 Summary Results for *Small Remote* Database (NLOR)

	Lookup		Traversal		Insert		L+T+I	
Benchmark	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	68.093	33.261	126.904	63.245	91.428	62.962	286.425	159.468
2	167.569	90.903	113.670	60.331	88.838	62.542	370.077	213.776
3	164.442	90.738	114.286	60.890	91.506	61.521	370.234	213.149
4	167.587	91.513	125.116	68.061	90.471	61.048	383.174	220.622
5	163.239	88.358	124.105	65.831	79.455	59.705	366.799	213.894
Average:	146.186	78.955	120.816	63.672	88.340	61.556	355.342	204.182
Sample STD:	43.697	25.572	6.326	3.279	5.082	1.288	39.033	25.181

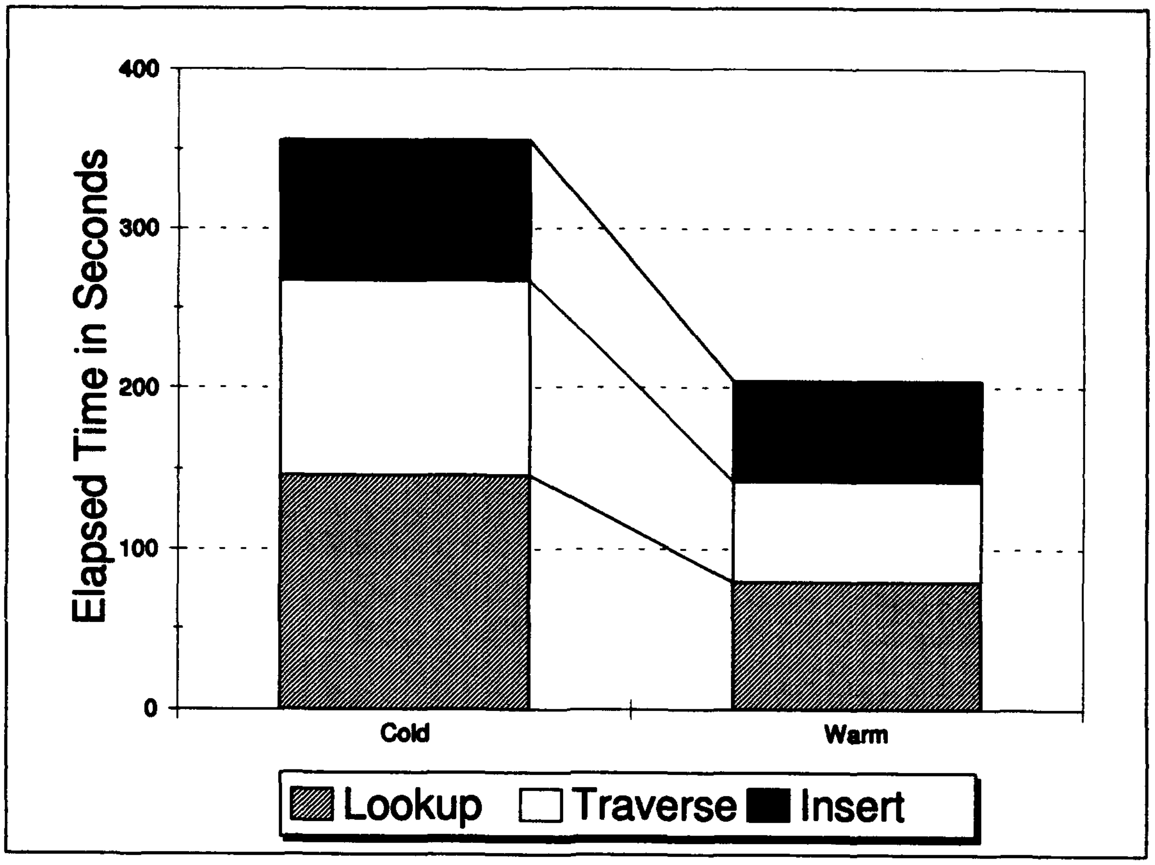

Figure 44. Matisse OO1 Average Benchmark Results for *Small Remote* Database (NLOR)

Table 40. Matisse OO1 Normalized Reverse Traversal Results for *Small Remote* Database (NLOR)

Benchmark	Cold	Warm
1	112.404	148.988
2	228.861	137.388
3	1256.711	83.885
4	113.683	71.195
5	151.021	71.305
Average:	372.536	102.552
Sample STD:	496.526	37.676

Table 41. Matisse OO1 Raw Benchmark Results for *Small Remote* Database (NLOR)

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	68.093	39.950	34.586	32.998	32.149	32.579	32.218	31.449	31.651	31.767
	Traversal	126.904	86.690	64.786	60.695	59.510	60.111	59.797	59.516	59.668	58.433
	Insert	91.428	80.394	72.107	63.242	60.103	60.232	57.787	57.473	57.057	58.266
	R. Traversal	366.375	11.348	59.635	120.281	75.593	54.744	81.852	98.741	0.226	2.312
2	Lookup	167.569	139.131	127.523	117.153	121.891	127.354	88.547	32.451	32.240	31.835
	Traversal	113.670	66.320	61.781	59.751	59.445	59.738	58.606	58.900	58.657	59.781
	Insert	88.838	78.231	68.020	62.992	60.483	59.137	57.401	57.504	61.360	57.753
	R. Traversal	46.540	151.188	0.180	11.456	152.700	53.462	66.299	74.448	140.452	21.837
3	Lookup	164.442	143.997	125.967	117.903	125.820	122.932	85.191	31.506	31.828	31.494
	Traversal	114.286	67.844	61.410	60.664	59.669	60.864	60.068	59.000	59.875	58.612
	Insert	91.506	77.463	67.755	61.347	58.053	59.482	59.436	57.799	57.209	55.144
	R. Traversal	0.766	121.011	137.003	115.912	52.922	115.627	75.928	50.584	100.121	15.288
4	Lookup	167.587	130.123	123.908	139.042	127.156	126.021	80.841	31.926	32.438	32.165
	Traversal	125.116	92.236	88.745	67.280	60.928	61.188	60.228	60.520	60.869	60.557
	Insert	90.471	70.394	64.939	62.427	63.127	58.750	56.403	57.453	57.751	58.186
	R. Traversal	160.924	102.131	69.476	54.572	75.048	72.371	51.708	33.104	19.611	111.181
5	Lookup	163.239	110.777	126.521	120.944	122.854	124.310	95.242	31.469	31.657	31.451
	Traversal	124.105	96.455	76.426	62.388	60.603	60.419	58.937	58.963	60.094	58.189
	Insert	79.455	65.938	64.789	60.855	58.735	57.156	57.917	56.075	57.702	58.175
	R. Traversal	77.076	104.483	15.849	92.541	66.057	39.133	72.587	125.049	82.020	130.764

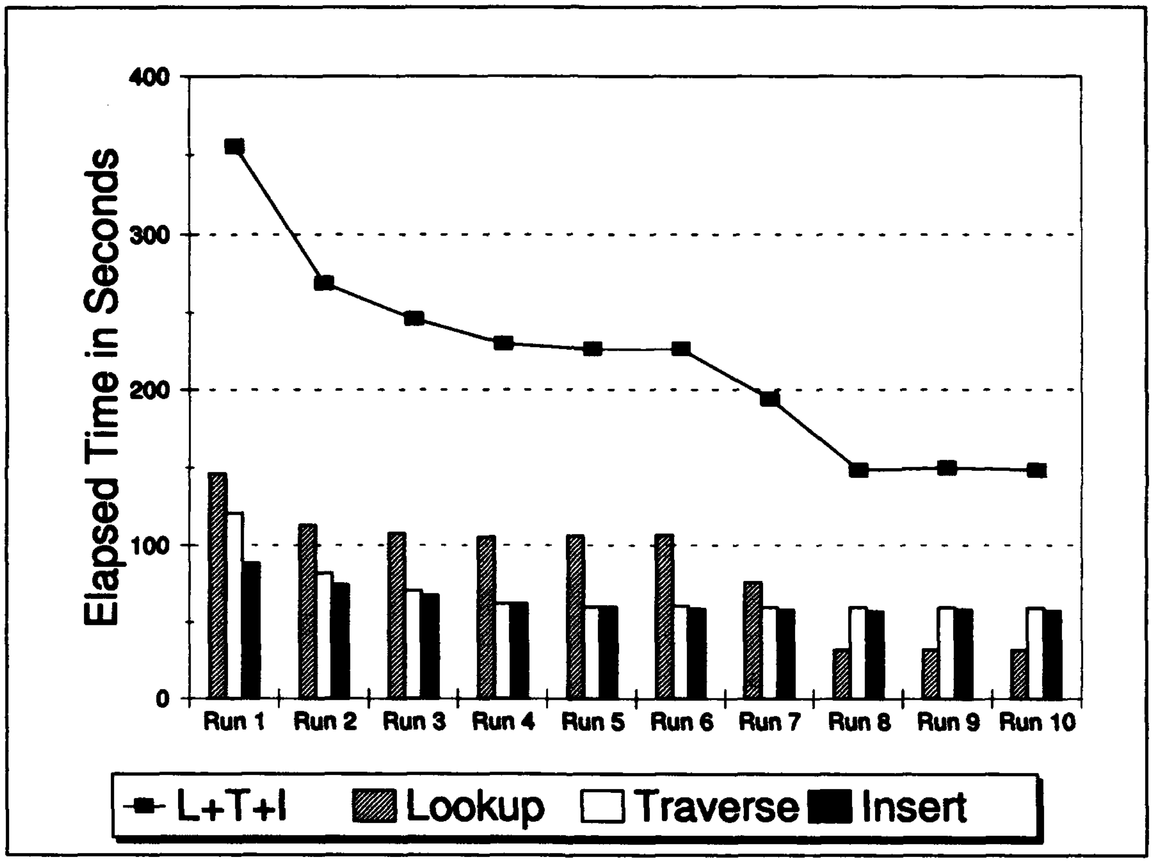

Figure 45. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Remote* Database (NLOR)

Table 42. Matisse OO1 Summary Results for *Small Remote* Database (NLOR) 200 Second Delay for Version Collection

	Lookup		Traversal		Insert		L+T+I	
Benchmark	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	58.049	33.851	118.312	63.674	88.274	59.917	264.635	157.442
2	131.126	73.193	126.512	63.781	83.832	62.643	341.470	199.617
3	162.249	76.295	126.278	63.519	94.277	63.681	382.804	203.495
4	169.497	77.252	115.130	62.263	92.898	63.306	377.525	202.821
5	162.869	77.185	91.375	62.593	91.670	62.400	345.914	202.178
Average:	136.758	67.555	115.521	63.166	90.190	62.389	342.470	193.111
Sample STD:	46.450	18.914	14.383	0.690	4.193	1.474	47.241	19.993

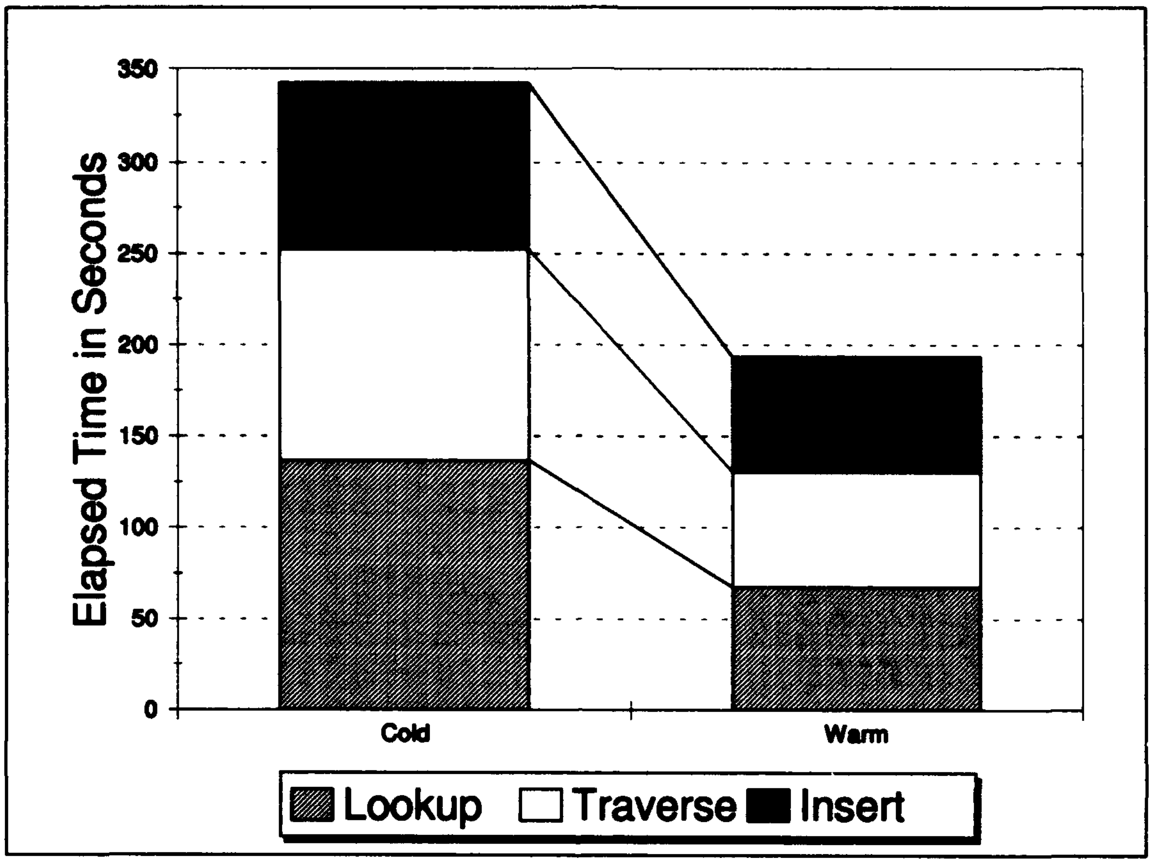

Figure 46. Matisse OO1 Average Benchmark Results for *Small Remote* Database (NLOR) 200 Second Delay for Version Collection

Table 43. Matisse OO1 Normalized Reverse Traversal Results for *Small Remote* Database (NLOR) 200 Second Delay for Version Collection

Benchmark	Cold	Warm
1	108.797	149.511
2	231.839	133.387
3	2148.554	88.179
4	143.861	74.118
5	162.090	75.978
Average:	559.028	104.235
Sample STD:	889.702	34.868

Table 44. Matisse OO1 Raw Benchmark Results for *Small Remote* Database (NLOR) 200 Second Delay for Version Collection

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	58.049	37.736	35.024	33.847	33.163	33.266	33.706	32.577	32.639	32.700
	Traversal	118.312	76.224	64.016	62.402	62.206	62.355	62.034	61.330	61.831	60.665
	Insert	88.274	72.369	59.409	56.600	57.064	57.015	54.987	56.231	61.689	63.884
	R. Traversal	354.619	10.473	57.729	121.648	74.701	53.085	81.331	99.227	0.231	2.260
2	Lookup	131.126	66.177	90.003	75.635	118.144	125.067	85.185	32.639	32.933	32.952
	Traversal	126.512	75.063	64.746	62.572	61.838	62.711	61.494	61.694	61.356	62.557
	Insert	83.832	70.440	66.882	62.871	60.322	60.595	65.076	60.343	57.763	59.495
	R. Traversal	47.145	154.685	0.150	15.998	172.790	55.641	68.312	76.688	145.188	22.382
3	Lookup	162.249	131.110	118.810	121.058	119.806	64.528	33.298	32.561	32.735	32.754
	Traversal	126.278	75.560	64.264	62.447	61.564	62.681	61.592	61.084	61.999	60.477
	Insert	94.277	79.453	70.690	63.936	62.837	59.146	58.330	60.174	60.194	58.365
	R. Traversal	1.310	124.494	158.766	121.253	54.782	119.153	78.420	52.490	103.506	15.966
4	Lookup	169.497	118.059	109.876	112.558	108.738	114.136	33.754	32.428	32.907	32.813
	Traversal	115.130	66.377	62.910	61.988	60.643	61.905	61.312	61.522	62.043	61.669
	Insert	92.898	77.841	71.939	66.563	61.136	59.559	60.108	57.349	57.576	57.682
	R. Traversal	203.642	110.848	71.802	56.887	77.638	75.097	53.548	34.291	20.193	115.143
5	Lookup	162.869	116.504	109.002	113.402	112.479	112.042	32.688	32.943	32.820	32.785
	Traversal	91.375	66.116	62.826	62.412	62.570	62.630	61.576	61.713	62.797	60.697
	Insert	91.670	73.449	64.750	65.052	60.256	58.659	57.395	59.279	64.846	57.914
	R. Traversal	82.725	135.462	16.413	96.144	68.633	40.119	75.178	129.136	84.824	135.260

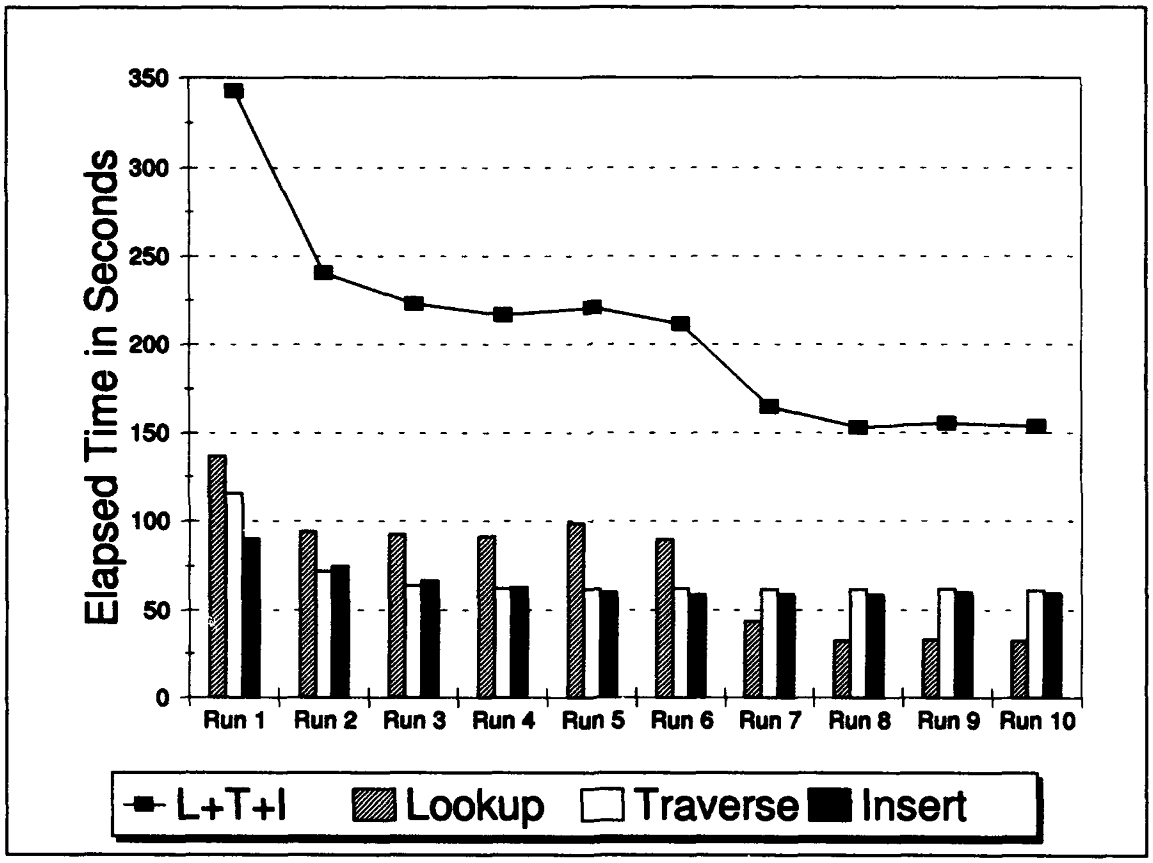

Figure 47. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Remote* Database (NLOR) 200 Second Delay for Version Collection

Table 45. Matisse OO1 Summary Results for *Small Local* Database (NLOR)

	Lookup		Traversal		Insert		L+T+I	
Benchmark	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	45.433	27.340	94.924	54.260	80.878	57.134	221.235	138.734
2	98.699	31.953	133.009	62.135	126.723	62.028	358.431	156.116
3	143.486	32.782	114.513	52.302	90.308	63.489	348.307	148.573
4	132.914	30.563	184.515	60.324	97.573	63.690	415.002	154.577
5	119.174	36.256	201.281	61.896	96.282	63.041	416.737	161.193
Average:	107.941	31.779	145.648	58.183	98.353	61.876	351.942	151.839
Sample STD:	38.749	3.251	45.573	4.582	17.174	2.728	79.553	8.598

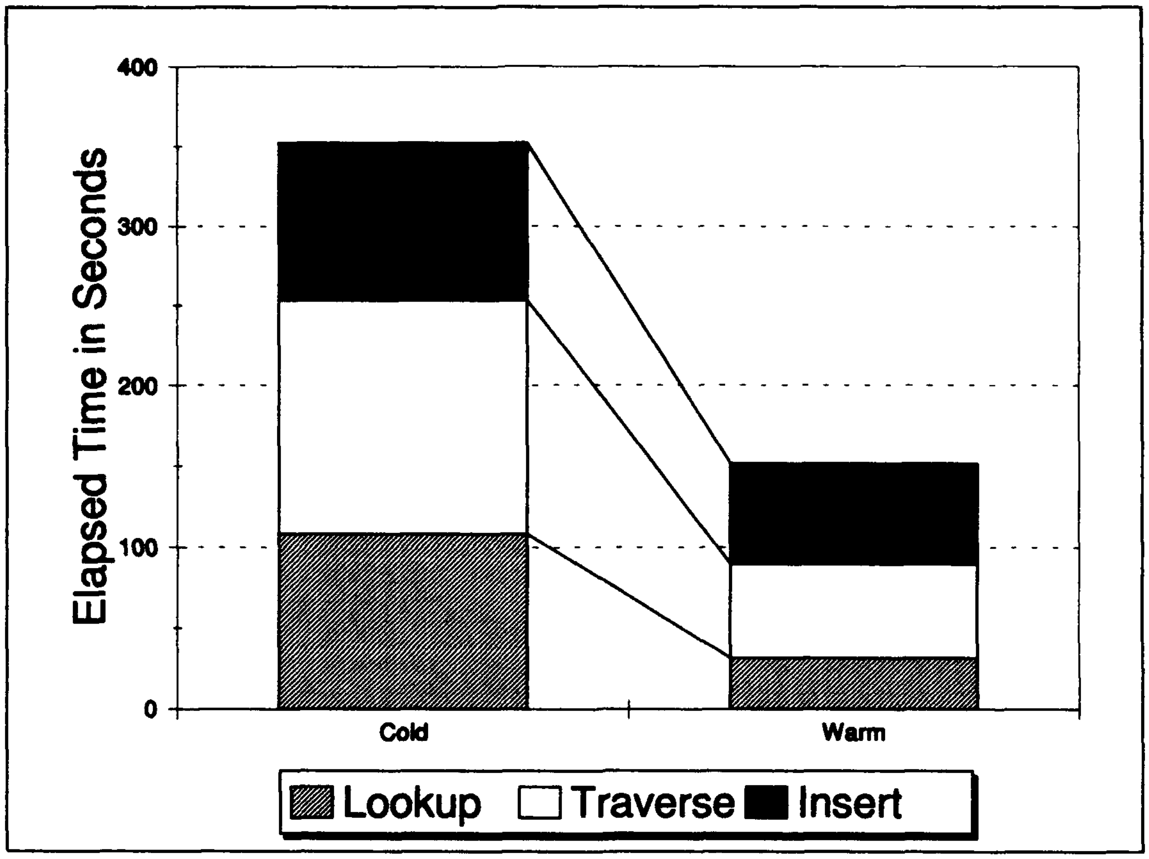

Figure 48. Matisse OO1 Average Benchmark Results for *Small Local* Database (NLOR)

Table 46. Matisse OO1 Normalized Reverse Traversal Results for *Small Local* Database (NLOR)

Benchmark	Cold	Warm
1	89.274	129.500
2	343.966	242.441
3	1869.751	96.081
4	267.905	63.348
5	378.278	76.351
Average:	589.835	121.544
Sample STD:	724.153	72.036

Table 47. Matisse OO1 Raw Benchmark Results for *Small Local* Database (NLOR)

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	45.433	28.048	27.272	27.346	27.106	26.984	27.469	27.067	27.399	27.365
	Traversal	94.924	73.101	69.559	55.075	49.403	48.895	48.669	47.903	48.399	47.334
	Insert	80.878	61.728	56.602	57.633	57.332	56.454	55.457	55.781	56.452	56.767
	R. Traversal	290.985	8.923	45.917	97.000	60.248	43.176	65.878	79.411	0.210	1.865
2	Lookup	98.699	59.076	32.579	28.841	28.514	27.920	27.926	27.775	27.474	27.469
	Traversal	133.009	137.290	71.975	53.994	48.384	50.519	48.182	49.708	48.847	50.320
	Insert	126.723	84.599	65.373	61.333	58.824	58.401	57.901	57.766	57.434	56.621
	R. Traversal	69.947	110.221	0.501	9.011	126.097	48.118	57.106	63.087	119.714	21.082
3	Lookup	143.486	59.508	40.138	30.034	28.086	27.605	27.792	27.340	27.154	27.379
	Traversal	114.513	64.440	52.221	55.092	49.539	51.101	49.079	48.941	50.046	50.262
	Insert	90.308	76.266	65.536	63.906	62.131	60.189	59.419	61.294	61.460	61.198
	R. Traversal	1.140	230.056	149.785	100.088	48.137	98.495	67.353	45.681	85.303	15.373
4	Lookup	132.914	46.690	31.588	28.818	28.105	27.986	27.892	28.349	28.057	27.584
	Traversal	184.515	117.408	65.528	52.657	50.781	51.026	50.519	51.207	52.440	51.351
	Insert	97.573	81.450	69.414	63.205	59.890	59.968	62.465	59.132	57.766	59.918
	R. Traversal	379.232	100.925	61.074	48.991	64.380	62.804	46.164	28.834	17.636	94.869
5	Lookup	119.174	50.037	60.267	39.374	32.425	30.137	28.158	27.811	30.278	27.813
	Traversal	201.281	132.987	61.277	54.097	53.336	51.805	50.600	51.232	51.594	50.136
	Insert	96.282	75.070	66.137	63.808	65.018	60.647	59.506	59.710	58.555	58.922
	R. Traversal	193.060	240.426	19.258	81.283	57.168	33.131	61.764	106.994	69.890	112.089

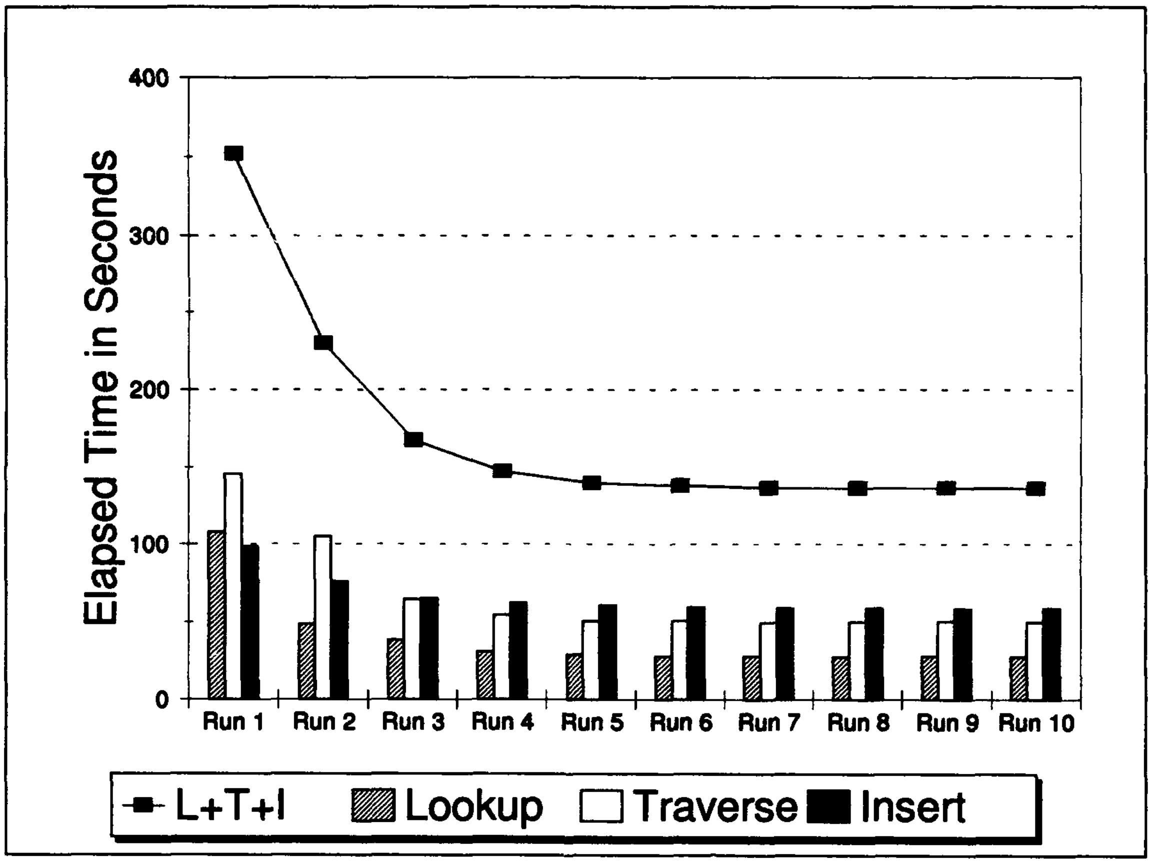

Figure 49. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database (NLOR)

Table 48. Matisse OO1 Summary Results for *Large Remote* Database

	Lookup		Traversal		Insert		L+T+I	
Benchmark	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	188.324	175.267	226.445	207.408	116.854	106.712	531.623	489.387
2	264.613	241.197	226.844	202.153	126.629	111.419	618.086	554.769
3	263.356	222.336	200.613	221.924	115.350	112.770	579.319	557.030
5	266.949	226.600	192.983	207.887	120.863	108.073	580.795	542.560
Average:	245.811	216.350	231.703	214.405	119.756	110.767	577.456	535.937
Sample STD:	38.353	28.555	47.183	12.560	4.380	3.350	35.431	31.677

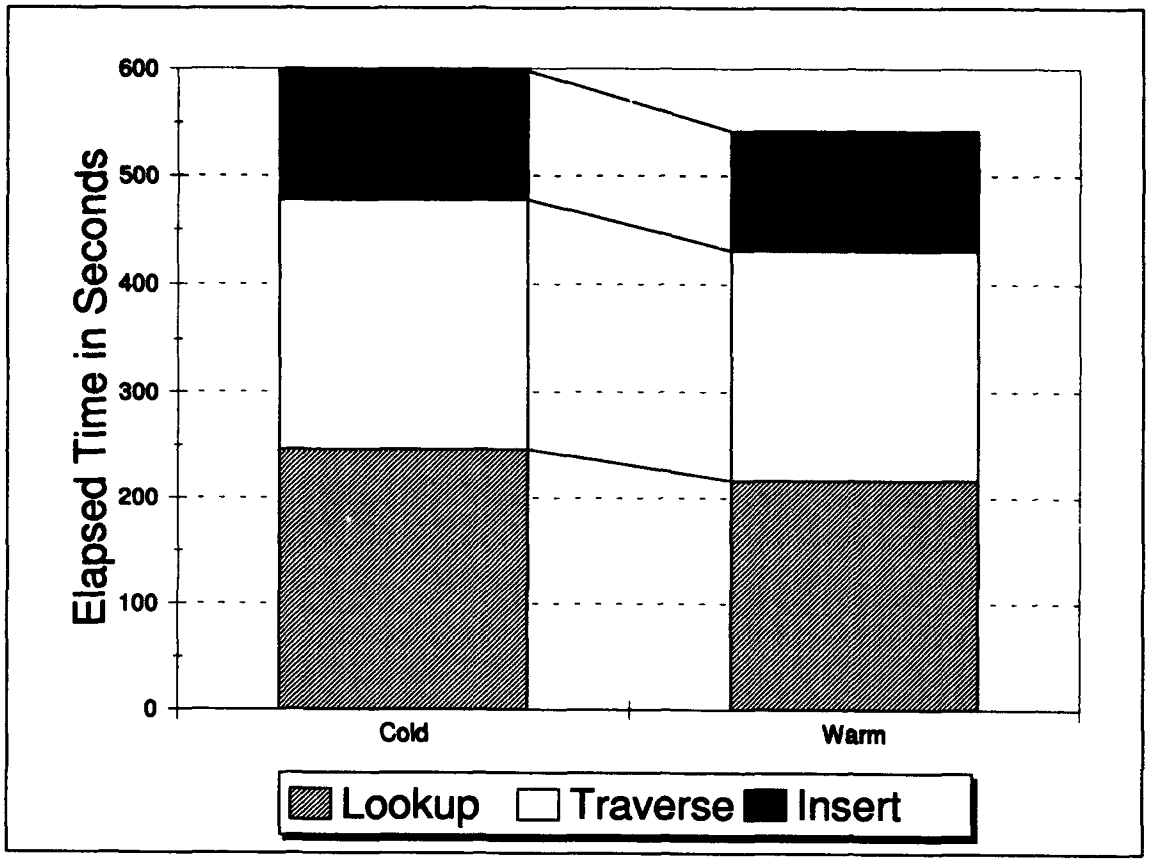

Figure 50. Matisse OO1 Average Benchmark Results for *Large Remote* Database

Table 49. Matisse OO1 Normalized Reverse Traversal Results for *Large Remote* Database

Benchmark	Cold	Warm
2	282.248	502.245
3	306.191	268.199
5	243.775	626.545
Average:	277.405	465.663
Sample STD:	31.489	181.952

Table 50. Matisse OO1 Raw Benchmark Results for *Large Remote* Database

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	188.324	173.870	173.517	180.518	171.316	178.382	173.217	176.781	177.082	172.718
	Traversal	226.445	215.646	175.123	223.261	217.513	197.556	195.704	211.625	213.195	217.051
	Insert	116.854	92.304	97.246	100.953	110.217	109.203	105.574	110.356	122.343	112.212
2	Lookup	264.613	243.012	237.361	244.284	249.484	254.333	248.903	228.795	232.411	232.187
	Traversal	226.844	194.956	199.838	220.079	212.264	183.528	198.512	191.735	225.066	193.402
	Insert	126.629	115.772	105.187	106.221	107.898	108.559	117.297	111.883	115.731	114.224
	R. Traversal	407.968	276.668	50.921	83.741	638.010	213.178	544.933	134.304	258.600	0.727
3	Lookup	263.356	246.869	244.971	249.401	239.037	242.459	249.847	177.827	175.984	174.631
	Traversal	200.613	197.209	242.111	220.253	220.341	212.701	213.651	215.184	242.836	233.034
	Insert	115.350	112.942	106.757	106.610	113.445	116.724	125.036	119.035	103.397	110.980
	R. Traversal	82.896	498.514	238.497	45.482	43.343	490.442	327.393	600.917	245.992	301.219
4	Traversal	311.628	285.688	288.460	227.549	164.977	258.301	193.027	212.766	244.610	218.485
	Insert	119.084	111.170	118.498	115.913	113.599	109.937	120.033	118.522	106.451	119.617
5	Lookup	266.949	249.027	246.521	262.391	244.201	252.710	253.970	177.029	176.064	177.484
	Traversal	192.983	175.866	207.874	209.562	194.629	213.420	213.184	213.664	226.667	216.116
	Insert	120.863	108.783	102.066	103.015	101.835	111.815	114.208	110.768	111.246	108.919
	R. Traversal	188.628	129.481	195.547	241.633	356.943	269.928	263.722	329.405	147.214	1.073

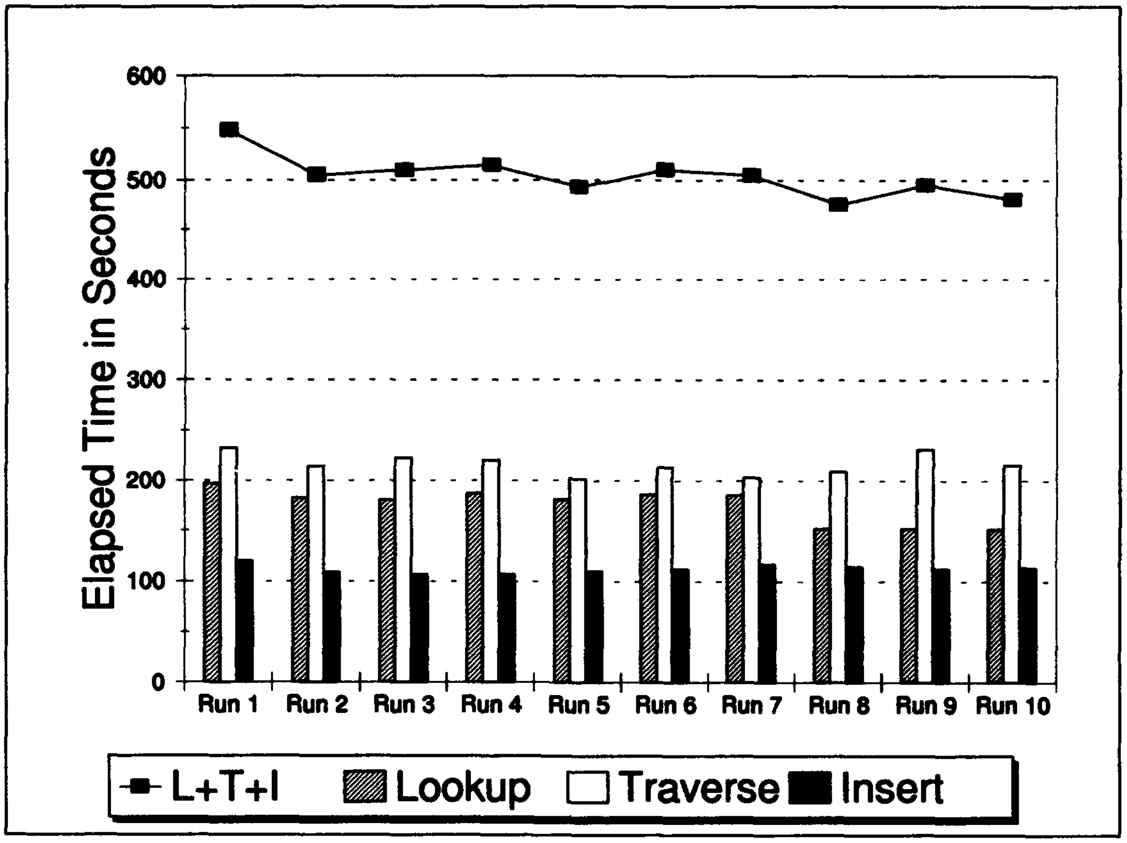

Figure 51. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Large Remote* Database

Table 51. Matisse OO1 Summary Results for *Large Local* Database

Benchmark	Lookup		Traversal		Insert		L+T+I	
	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	194.644	181.786	229.244	224.510	104.349	112.823	528.237	519.119
2	241.965	226.535	231.668	216.484	103.761	114.324	577.394	557.343
3	237.328	225.357	198.680	239.218	110.754	112.142	546.762	576.717
5	238.384	226.532	191.862	218.290	104.087	113.087	534.333	557.909
Average:	228.080	215.053	229.644	228.602	106.308	112.559	546.682	552.772
Sample STD:	22.379	22.185	41.520	12.606	3.171	1.433	21.878	24.174

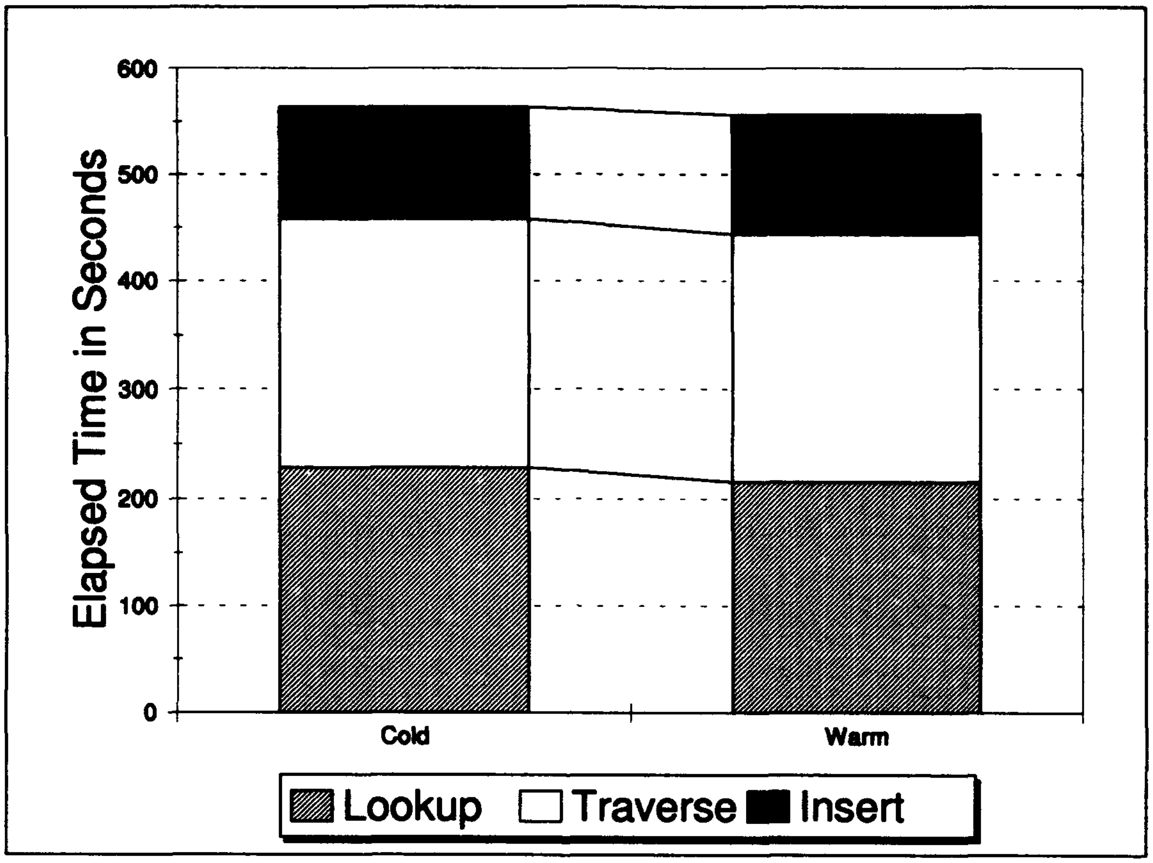

Figure 52. Matisse OO1 Average Benchmark Results for *Large Local* Database

Table 52. Matisse OO1 Normalized Reverse Traversal Results for *Large Local* Database

Benchmark	Cold	Warm
2	297.678	921.744
3	298.225	301.901
5	247.150	560.243
Average:	281.018	594.629
Sample STD:	29.332	311.349

Table 53. Matisse OO1 Raw Benchmark Results for *Large Local* Database

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	194.644	180.892	185.116	188.225	178.978	184.577	176.918	180.816	181.338	179.216
	Traversal	229.244	230.973	190.065	241.214	233.779	211.868	211.614	230.521	237.324	233.234
	Insert	104.349	104.441	107.890	112.129	118.602	114.683	114.548	113.705	118.898	110.517
2	Lookup	241.965	221.211	247.769	242.888	235.491	217.311	214.922	226.942	231.976	200.308
	Traversal	231.668	206.596	215.092	231.835	230.135	199.068	211.857	205.257	236.279	212.235
	Insert	103.761	105.760	112.951	113.381	117.537	115.181	118.367	116.245	115.494	114.002
	R. Traversal	430.271	298.765	53.347	89.286	731.071	252.524	647.848	161.289	293.048	1.794
3	Lookup	237.328	245.226	234.604	231.607	235.474	212.068	213.073	229.874	232.868	193.423
	Traversal	198.680	212.140	260.038	241.382	238.897	232.257	233.501	232.535	254.324	247.890
	Insert	110.754	115.566	107.657	111.027	116.230	109.910	111.646	109.168	112.527	115.548
	R. Traversal	80.739	539.383	270.879	51.603	43.942	553.936	378.346	710.038	285.713	345.201
4	Traversal	296.764	274.969	272.591	282.649	187.064	254.435	201.212	229.804	259.120	238.736
	Insert	108.588	102.189	115.303	115.664	125.226	115.040	107.041	101.339	103.189	108.792
5	Lookup	238.384	245.090	247.698	230.181	237.424	215.187	232.104	211.614	239.767	179.720
	Traversal	191.862	184.948	213.960	216.924	202.567	222.994	225.367	227.804	240.336	229.714
	Insert	104.087	102.079	108.262	113.779	120.028	119.142	137.220	109.634	104.896	102.749
	R. Traversal	191.240	131.357	199.529	251.997	380.559	298.092	280.172	355.558	160.655	0.853

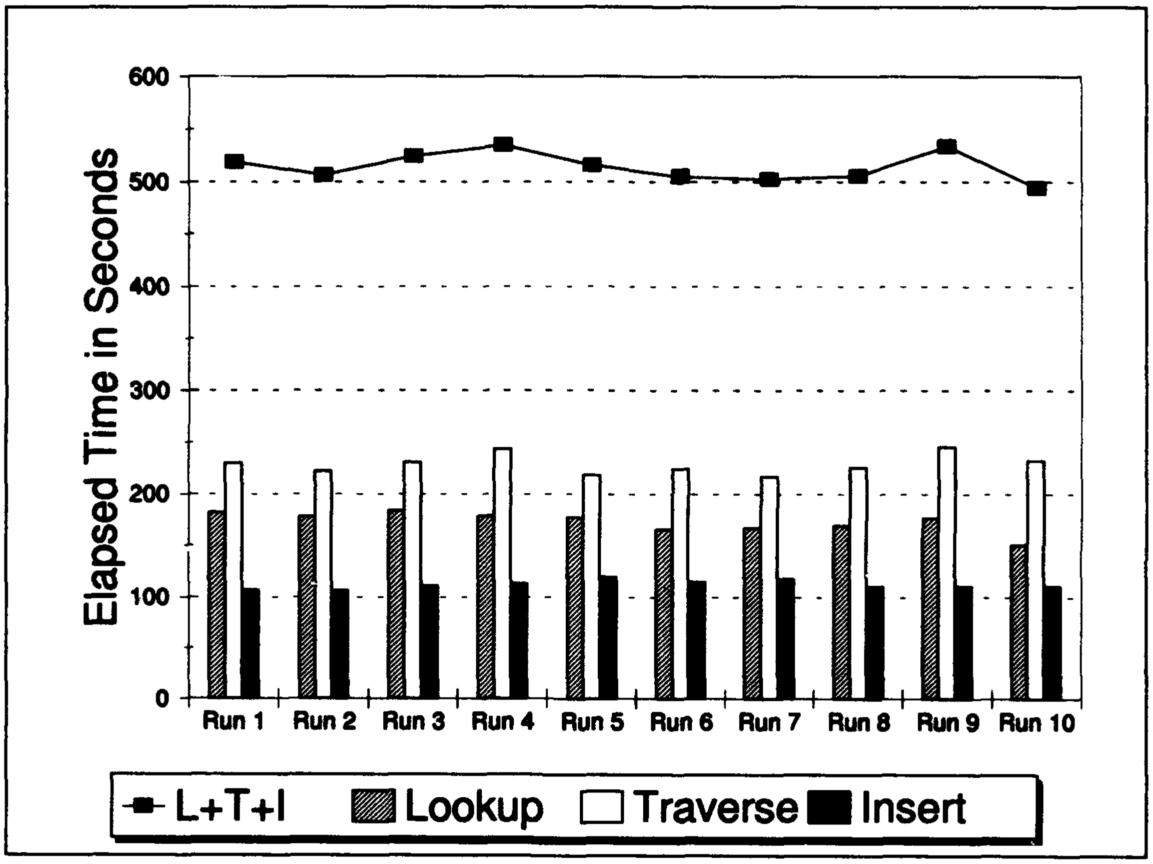

Figure 53. Matisse OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Large Local* Database

B.3 OO1 Benchmark Results for the ObjectStore DBMS

This section reports our OO1 benchmark results for the ObjectStore DBMS. Results for the following database configurations are provided:

- ObjectStore *Small Remote* Database — The results for this benchmark configuration are reported in Tables 54, 55, and 56 and in Figures 54 and 55.
- ObjectStore *Small Local* Database — The results for this benchmark configuration are reported in Tables 57, 58, and 59 and in Figures 56 and 57.
- ObjectStore *Small Local* Database (second complete run) — The results for this benchmark configuration are reported in Tables 60, 61, and 62 and in Figures 58 and 59.
- ObjectStore *Small Remote* Database with No Locality of Reference (NLOR) — The results for this benchmark configuration are reported in Tables 63, 64, and 65 and in Figures 60 and 61.
- ObjectStore *Small Local* Database with No Locality of Reference — The results for this benchmark configuration are reported in Tables 66, 67, and 68 and in Figures 62 and 63.
- ObjectStore *Large Remote* Database — The results for this benchmark configuration are reported in Tables 69, 70, and 71 and in Figures 64 and 65.
- ObjectStore *Large Local* Database — The results for this benchmark configuration are reported in Tables 72, 73, and 74 and in Figures 66 and 67.

Table 54. ObjectStore OO1 Summary Results for *Small Remote* Database

	Lookup		Traversal		Insert		L+T+I	
Benchmark	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	23.650	1.102	21.354	1.621	16.480	2.794	61.484	5.517
2	29.546	1.200	29.761	1.596	17.525	3.249	76.832	6.045
3	28.251	1.244	27.230	1.616	15.229	3.502	70.710	6.362
4	28.795	1.241	27.802	1.916	16.912	3.657	73.509	6.814
5	30.712	1.408	25.463	1.921	15.280	2.719	71.455	6.048
Average:	28.191	1.239	26.322	1.734	16.285	3.184	70.798	6.157
Sample STD:	2.701	0.111	3.173	0.169	1.012	0.418	5.722	0.476

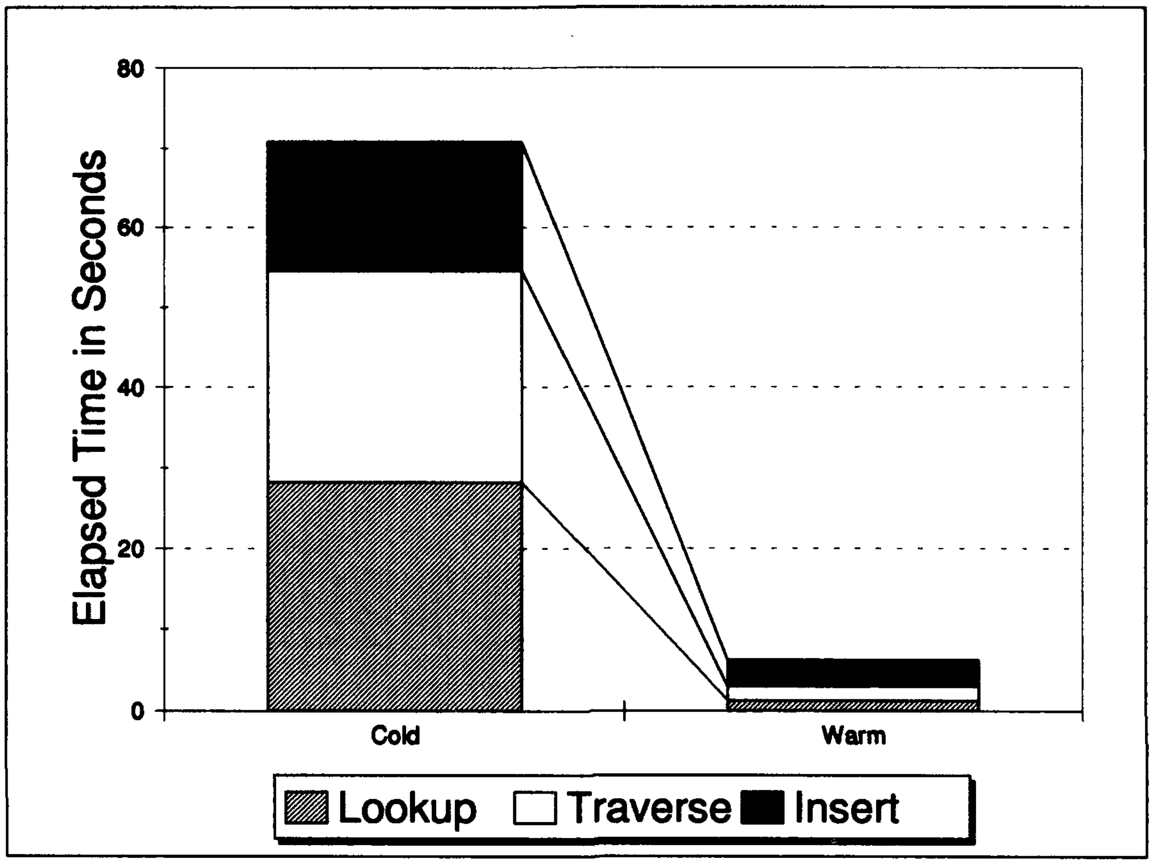

Figure 54. ObjectStore OO1 Average Benchmark Results for *Small Remote* Database

Table 55. ObjectStore OO1 Normalized Reverse Traversal Results for *Small Remote* Database

Benchmark	Cold	Warm
1	15.969	15.881
2	5809.241	2.794
3	11.617	1.993
4	17.484	1.842
5	22.339	1.409
Average:	1157.330	4.784
Sample STD:	2590.438	6.224

Table 56. ObjectStore OO1 Raw Benchmark Results for *Small Remote* Database

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	23.650	1.767	1.105	1.017	0.990	1.005	1.001	1.014	0.998	1.021
	Traversal	21.354	4.405	2.159	1.546	1.069	1.190	1.088	1.052	1.016	1.061
	Insert	16.480	3.340	3.487	2.740	2.230	2.473	2.054	1.990	2.174	4.657
	R. Traversal	29.606	1.948	0.040	2.509	1.398	1.247	0.845	1.360	1.457	1.218
2	Lookup	29.546	2.403	1.255	1.059	1.008	1.010	1.032	1.003	1.018	1.010
	Traversal	29.761	4.091	1.880	1.627	1.117	1.199	1.059	1.199	1.034	1.155
	Insert	17.525	3.939	3.136	2.873	2.540	2.606	4.297	2.056	2.089	5.707
	R. Traversal	1.771	36.574	1.854	1.726	1.928	1.787	0.584	1.083	1.333	1.198
3	Lookup	28.251	2.975	1.186	1.014	1.005	0.999	1.015	1.011	0.994	0.998
	Traversal	27.230	4.000	1.940	1.530	1.205	1.360	1.163	1.044	1.164	1.136
	Insert	15.229	3.707	7.058	3.441	2.206	2.525	2.091	6.190	2.359	1.938
	R. Traversal	20.005	5.575	2.672	1.329	2.426	3.275	2.176	2.092	1.768	2.127
4	Lookup	28.795	2.849	1.180	1.079	1.006	1.030	1.017	0.999	1.007	1.005
	Traversal	27.802	7.024	2.130	1.288	1.404	1.192	1.211	1.054	0.998	0.946
	Insert	16.912	3.796	4.196	3.939	2.764	2.808	2.742	2.340	2.106	8.224
	R. Traversal	23.022	4.080	1.764	1.338	1.179	1.207	1.137	0.796	1.483	0.131
5	Lookup	30.712	2.341	1.181	1.169	1.039	1.217	1.031	1.030	1.049	2.614
	Traversal	25.463	5.010	4.356	1.006	1.223	1.354	1.119	1.081	1.014	1.130
	Insert	15.280	3.372	3.491	2.857	2.524	2.427	2.474	3.023	2.223	2.076
	R. Traversal	19.615	4.835	1.153	2.018	1.217	1.172	1.147	1.437	1.537	0.329

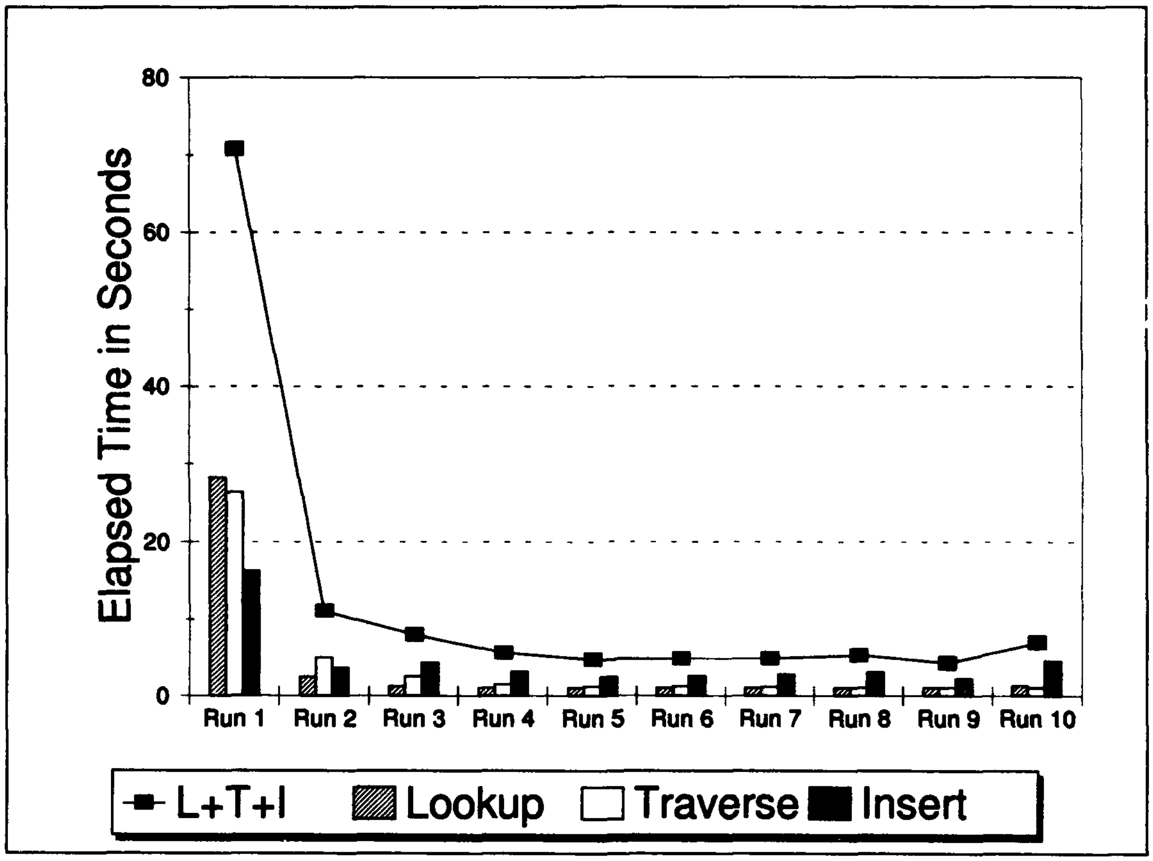

Figure 55. ObjectStore OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Remote* Database

Table 57. ObjectStore OO1 Summary Results for *Small Local* Database

	Lookup		Traversal		Insert		L+T+I	
Benchmark	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	21.486	1.090	30.618	2.386	21.603	3.036	73.707	6.512
2	29.071	1.182	31.976	2.313	18.527	4.066	79.574	7.561
3	27.759	1.532	30.584	2.385	18.505	4.144	76.848	8.061
4	28.410	1.206	18.208	1.489	17.338	2.793	63.956	5.488
5	27.563	1.484	17.838	1.861	18.272	4.605	63.673	7.950
Average:	26.858	1.299	25.845	2.087	18.849	3.729	71.552	7.114
Sample STD:	3.061	0.197	7.164	0.399	1.614	0.776	7.362	1.096

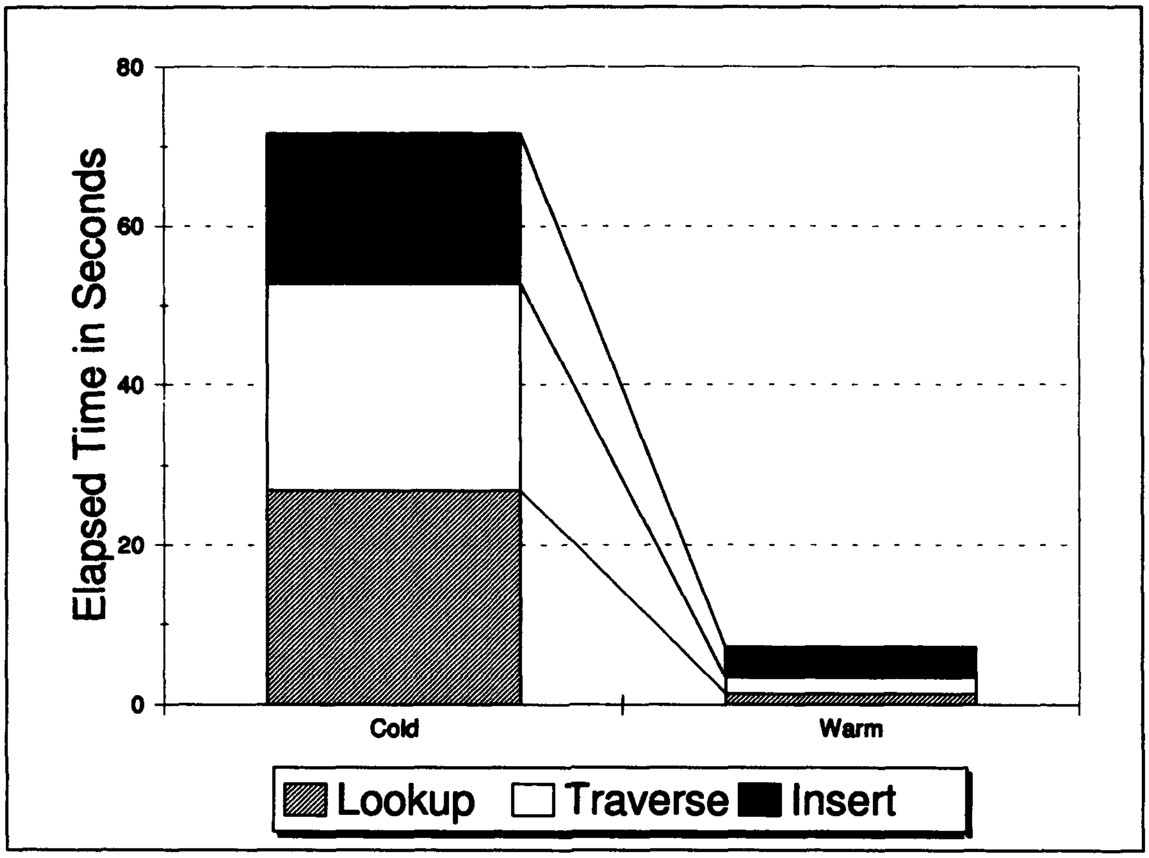

Figure 56. ObjectStore OO1 Average Benchmark Results for *Small Local* Database

Table 58. ObjectStore OO1 Normalized Reverse Traversal Results for *Small Local* Database

Benchmark	Cold	Warm
1	21.565	15.664
2	10433.572	2.835
3	22.718	1.291
4	26.808	2.140
5	37.703	1.912
Average:	2108.473	4.768
Sample STD:	4653.876	6.116

Table 59. ObjectStore OO1 Raw Benchmark Results for *Small Local* Database

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	21.486	1.716	1.087	0.996	0.970	0.985	1.003	1.003	1.035	1.016
	Traversal	30.618	9.907	2.738	1.794	1.041	1.731	1.161	1.028	1.028	1.046
	Insert	21.603	2.922	3.165	2.736	2.224	2.474	1.822	4.375	1.874	5.731
	R. Traversal	39.981	1.767	0.039	2.334	1.347	1.212	0.840	1.286	1.286	1.153
2	Lookup	29.071	2.477	1.253	0.985	0.982	0.978	1.004	0.977	0.990	0.987
	Traversal	31.976	6.340	2.391	2.117	1.148	1.239	1.031	4.393	1.018	1.140
	Insert	18.527	3.398	10.024	2.657	2.324	2.526	2.474	1.807	1.873	9.515
	R. Traversal	3.181	34.975	1.912	1.716	4.802	1.806	0.550	1.029	1.268	1.159
3	Lookup	27.759	3.031	3.794	1.005	0.993	0.992	1.000	1.000	0.983	0.988
	Traversal	30.584	9.511	2.720	1.745	1.650	1.397	1.143	1.044	1.137	1.119
	Insert	18.505	3.634	9.857	2.911	6.523	2.291	6.240	2.075	2.024	1.746
	R. Traversal	39.120	3.260	2.149	1.293	1.528	1.553	1.200	1.145	1.201	1.304
4	Lookup	28.410	2.696	1.167	1.053	0.986	1.008	0.995	0.978	0.989	0.987
	Traversal	18.208	3.820	1.871	1.225	1.328	1.135	1.155	1.030	0.943	0.896
	Insert	17.338	6.072	3.250	2.973	2.490	2.317	2.039	2.057	2.006	1.932
	R. Traversal	35.300	8.766	1.878	1.340	1.135	1.209	1.139	0.794	1.410	0.130
5	Lookup	27.563	5.033	1.163	1.140	0.991	1.002	1.019	1.014	1.006	0.992
	Traversal	17.838	3.993	1.593	0.943	1.158	1.209	4.575	1.188	0.996	1.092
	Insert	18.272	3.155	11.847	2.472	6.158	2.107	6.044	2.066	5.766	1.830
	R. Traversal	33.105	7.089	4.283	2.138	1.237	1.175	1.152	1.379	1.479	0.315

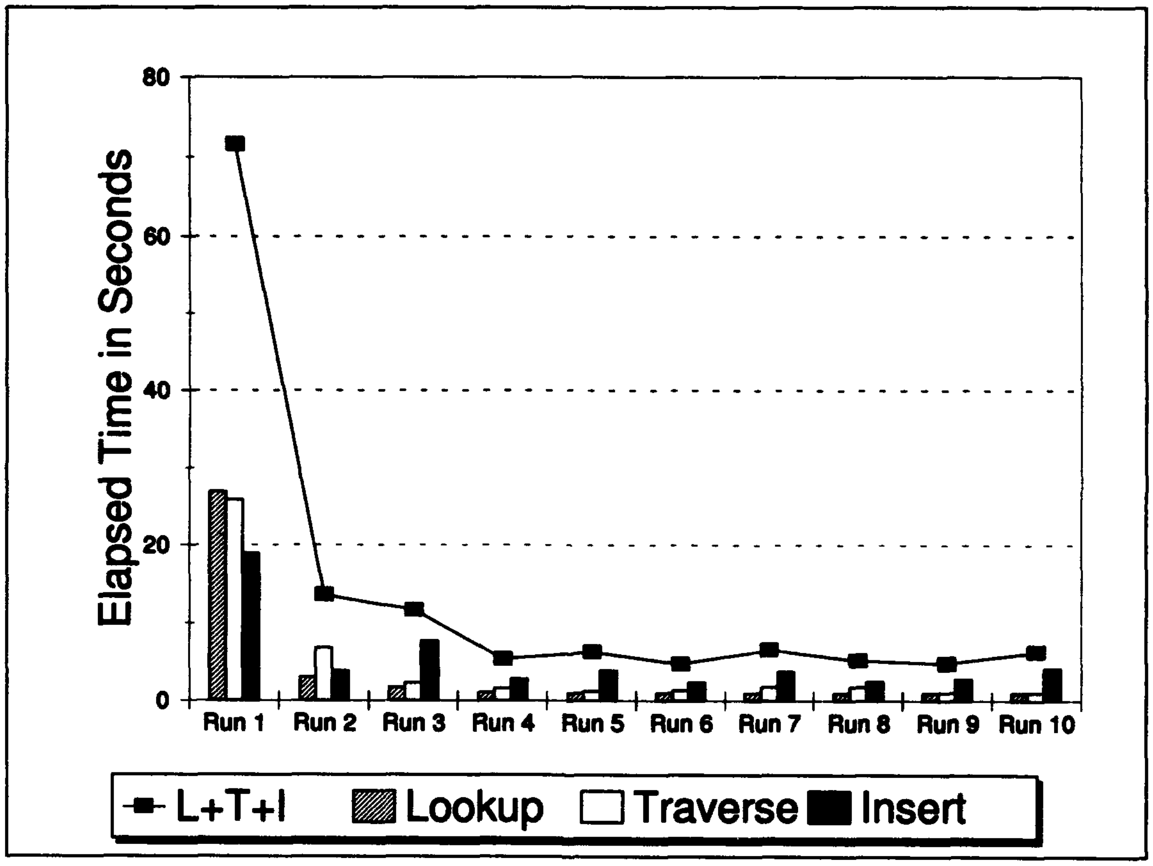

Figure 57. ObjectStore OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database

Table 60. ObjectStore OO1 Summary Results for *Small Local* Database (second)

	Lookup		Traversal		Insert		L+T+I	
Benchmark	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	30.260	1.191	30.444	2.372	16.288	2.529	76.992	6.092
2	28.647	1.212	19.009	1.484	18.672	3.760	66.328	6.456
3	29.491	1.257	31.736	2.386	16.887	4.059	78.114	7.702
4	28.064	1.222	31.207	2.453	19.805	2.811	79.076	6.486
5	27.833	1.489	32.750	2.323	17.305	3.159	77.888	6.971
Average:	28.859	1.274	29.029	2.204	17.791	3.264	75.680	6.741
Sample STD:	1.011	0.122	5.664	0.405	1.427	0.639	5.280	0.621

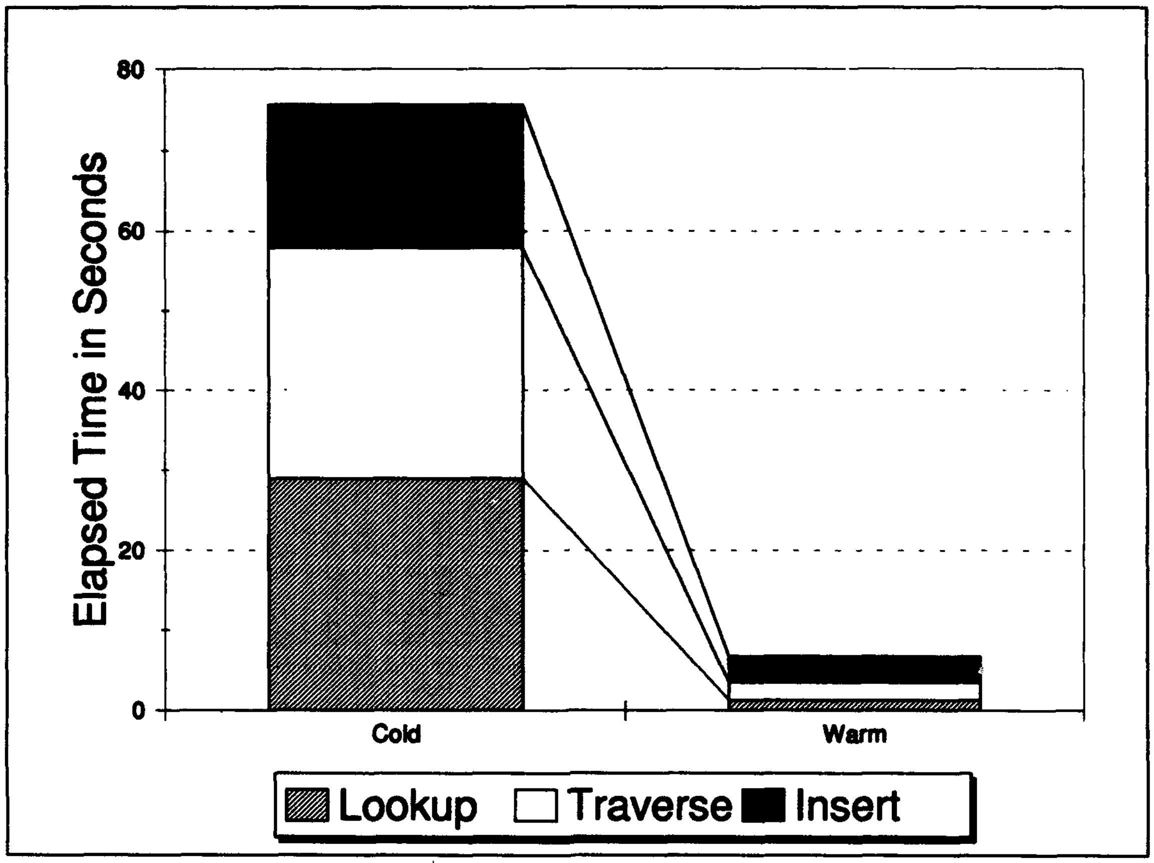

Figure 58. ObjectStore OO1 Average Benchmark Results for *Small Local* Database (second)

Table 61. ObjectStore OO1 Normalized Reverse Traversal Results for *Small Local* Database (second)

Benchmark	Cold	Warm
1	21.292	16.361
2	10177.650	2.803
3	21.662	1.434
4	27.452	2.146
5	39.282	1.859
Average:	2057.468	4.921
Sample STD:	4539.326	6.415

Table 62. ObjectStore OO1 Raw Benchmark Results for *Small Local* Database (second)

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	30.260	2.433	1.177	1.026	0.996	1.018	1.014	1.024	1.007	1.026
	Traversal	30.444	10.044	2.759	1.793	1.056	1.383	1.167	1.036	1.040	1.066
	Insert	16.288	2.528	2.623	2.156	2.346	2.013	1.755	1.713	1.890	5.739
	R. Traversal	39.475	1.934	0.041	2.499	1.380	1.234	0.871	1.308	1.289	1.187
2	Lookup	28.647	2.513	1.310	1.008	1.011	1.006	1.031	1.004	1.016	1.008
	Traversal	19.009	3.312	1.769	1.572	1.131	1.190	1.037	1.204	1.010	1.131
	Insert	18.672	3.061	11.917	2.624	2.040	2.228	2.189	1.840	1.922	6.023
	R. Traversal	3.103	36.827	1.976	1.740	1.887	1.857	0.559	1.062	1.302	1.195
3	Lookup	29.491	3.033	1.198	1.018	1.012	1.008	1.014	1.021	1.005	1.005
	Traversal	31.736	6.457	2.790	1.842	1.329	1.465	1.551	3.562	1.338	1.142
	Insert	16.887	6.856	9.122	2.842	2.767	2.093	1.956	6.456	2.114	2.323
	R. Traversal	37.301	3.423	2.085	1.228	1.557	4.393	1.218	1.162	1.225	1.346
4	Lookup	28.064	2.696	1.155	1.073	1.007	1.026	1.015	1.004	1.013	1.010
	Traversal	31.207	7.586	2.649	1.561	4.710	1.291	1.245	1.106	1.004	0.924
	Insert	19.805	3.156	3.316	2.958	2.748	2.240	1.972	2.074	2.023	4.815
	R. Traversal	36.147	5.457	4.987	1.449	1.175	1.269	1.236	0.779	1.460	0.137
5	Lookup	27.833	4.936	1.179	1.139	1.013	1.020	1.036	1.035	1.023	1.018
	Traversal	32.750	7.517	2.331	1.090	1.287	1.769	1.158	1.074	3.555	1.122
	Insert	17.305	3.138	2.951	2.372	2.294	5.876	2.106	5.973	2.033	1.689
	R. Traversal	34.491	7.222	1.307	2.226	1.249	3.855	1.324	1.418	1.516	0.324

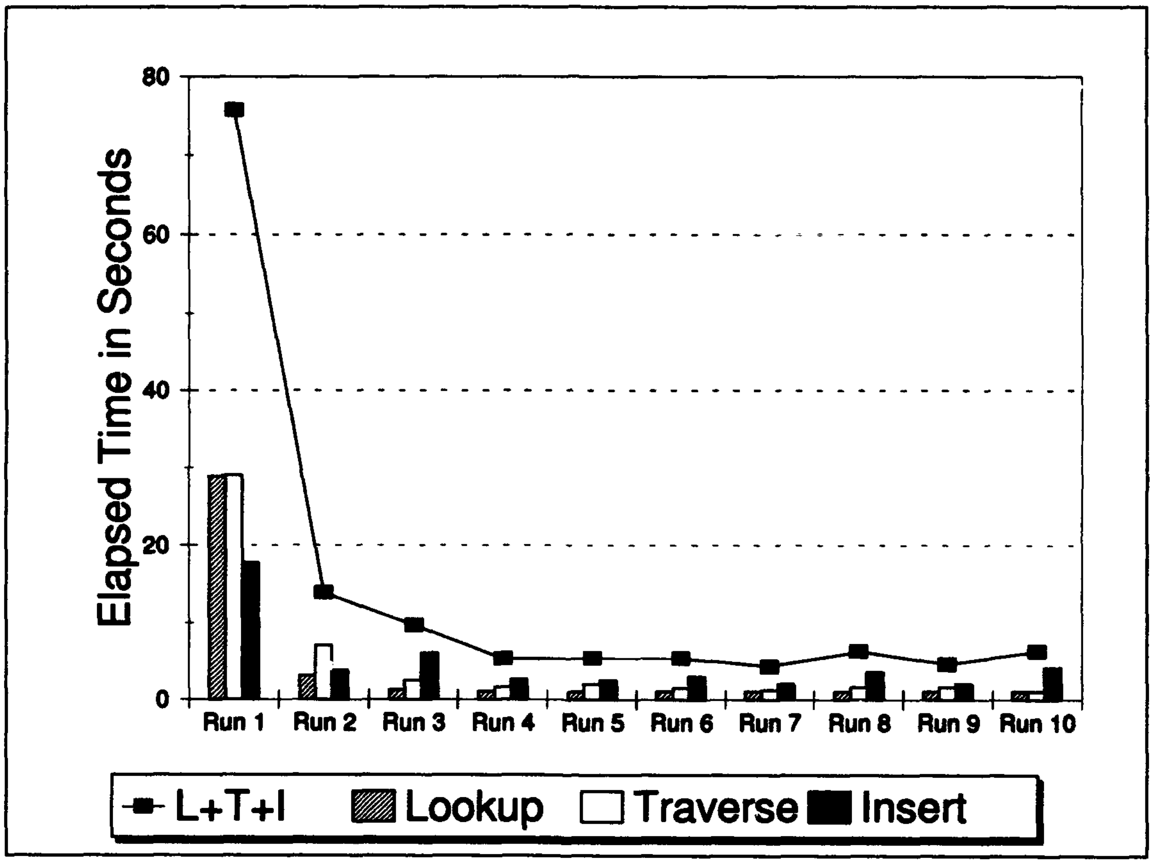

Figure 59. ObjectStore OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database (second)

Table 63. ObjectStore OO1 Summary Results for *Small Remote* Database (NLOR)

	Lookup		Traversal		Insert		L+T+I	
Benchmark	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	37.324	1.184	29.295	1.867	32.660	8.337	99.279	11.388
2	30.194	1.216	43.214	1.688	36.866	10.640	110.274	13.544
3	25.784	1.241	36.127	1.614	37.533	11.478	99.444	14.333
4	27.744	1.409	45.748	1.753	25.050	9.675	98.542	12.837
5	27.777	1.202	43.488	1.909	33.117	9.466	104.382	12.577
Average:	29.765	1.250	39.574	1.766	33.045	9.919	102.384	12.936
Sample STD:	4.506	0.091	6.787	0.122	4.970	1.196	4.982	1.102

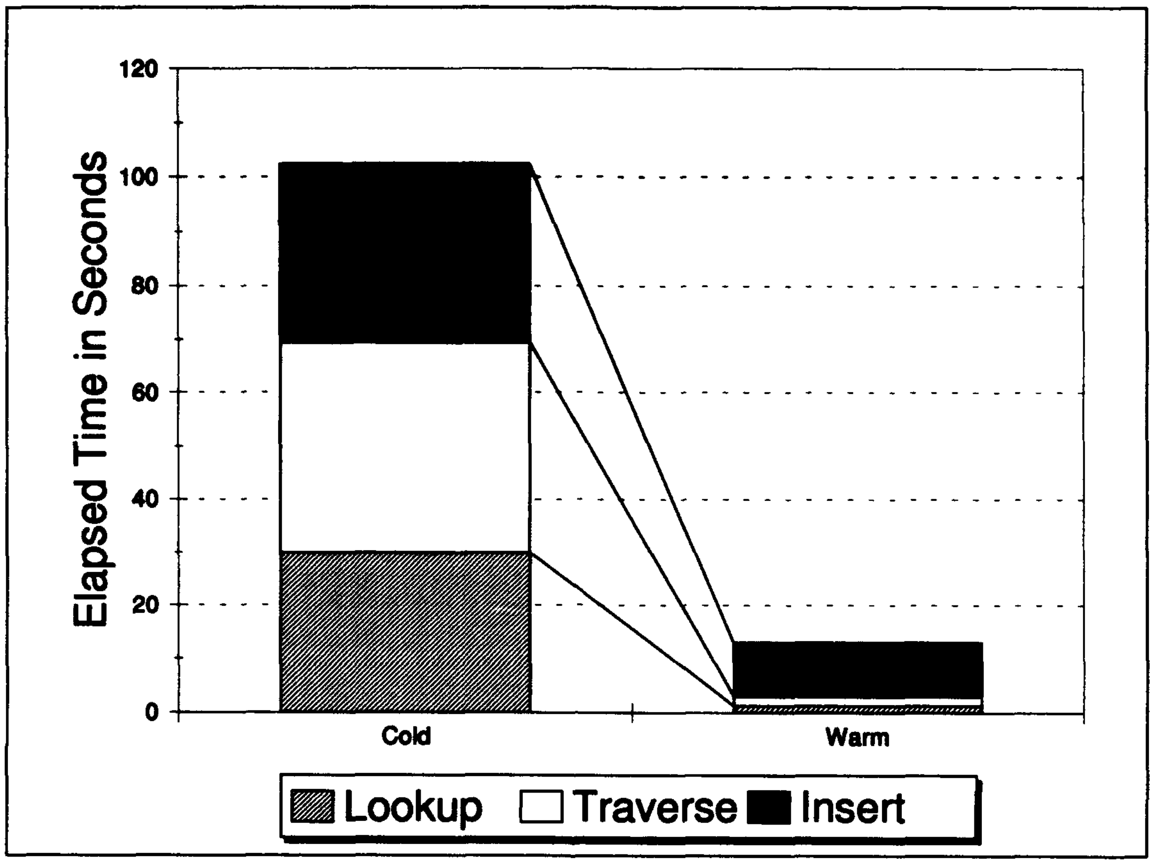

Figure 60. ObjectStore OO1 Average Benchmark Results for *Small Remote* Database (NLOR)

Table 64. ObjectStore OO1 Normalized Reverse Traversal Results for *Small Remote* Database (NLOR)

Benchmark	Cold	Warm
1	8.168	25.419
2	191.780	31.861
3	101.563	2.168
4	28.905	1.554
5	61.583	2.856
Average:	78.400	12.772
Sample STD:	72.555	14.671

Table 65. ObjectStore OO1 Raw Benchmark Results for *Small Remote* Database (NLOR)

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	37.324	2.425	1.150	1.019	0.990	1.011	1.011	1.020	1.007	1.020
	Traversal	29.295	4.279	1.592	1.602	1.543	1.531	1.547	1.602	1.551	1.561
	Insert	32.660	13.093	10.378	9.971	4.493	5.374	4.427	13.343	9.460	4.494
	R. Traversal	21.614	1.664	1.616	1.586	1.577	1.680	1.682	1.711	0.065	1.545
2	Lookup	30.194	2.570	1.307	1.009	1.002	1.004	1.030	1.000	1.014	1.005
	Traversal	43.214	2.249	1.646	1.556	1.608	1.567	1.603	1.583	1.804	1.572
	Insert	36.866	15.729	16.747	11.677	10.109	9.061	11.545	9.558	7.177	4.159
	R. Traversal	33.971	8.718	0.082	1.420	1.802	1.731	1.668	1.637	1.766	1.681
3	Lookup	25.784	2.890	1.202	1.019	1.015	1.002	1.019	1.012	1.003	1.007
	Traversal	36.127	1.950	1.605	1.560	1.562	1.563	1.566	1.550	1.620	1.548
	Insert	37.533	13.544	13.262	10.644	14.493	9.812	10.394	9.259	9.046	12.844
	R. Traversal	43.164	2.805	1.768	1.711	1.716	1.684	1.629	1.695	1.784	1.444
4	Lookup	27.744	2.601	2.935	1.080	0.999	1.027	1.015	1.003	1.011	1.005
	Traversal	45.748	2.374	1.593	1.593	1.559	1.603	1.906	2.038	1.572	1.541
	Insert	25.050	16.778	12.527	11.778	4.594	4.712	5.376	10.326	8.976	12.012
	R. Traversal	32.456	1.753	1.664	1.700	1.674	1.714	1.640	1.621	1.603	1.759
5	Lookup	27.777	2.359	1.197	1.154	1.011	1.012	1.032	1.029	1.018	1.003
	Traversal	43.488	1.878	1.583	1.554	1.545	4.083	1.888	1.569	1.518	1.561
	Insert	33.117	12.541	4.743	4.664	10.578	13.127	9.793	12.592	9.492	7.661
	R. Traversal	27.618	2.683	0.957	1.696	1.558	3.802	1.747	1.766	1.642	1.679

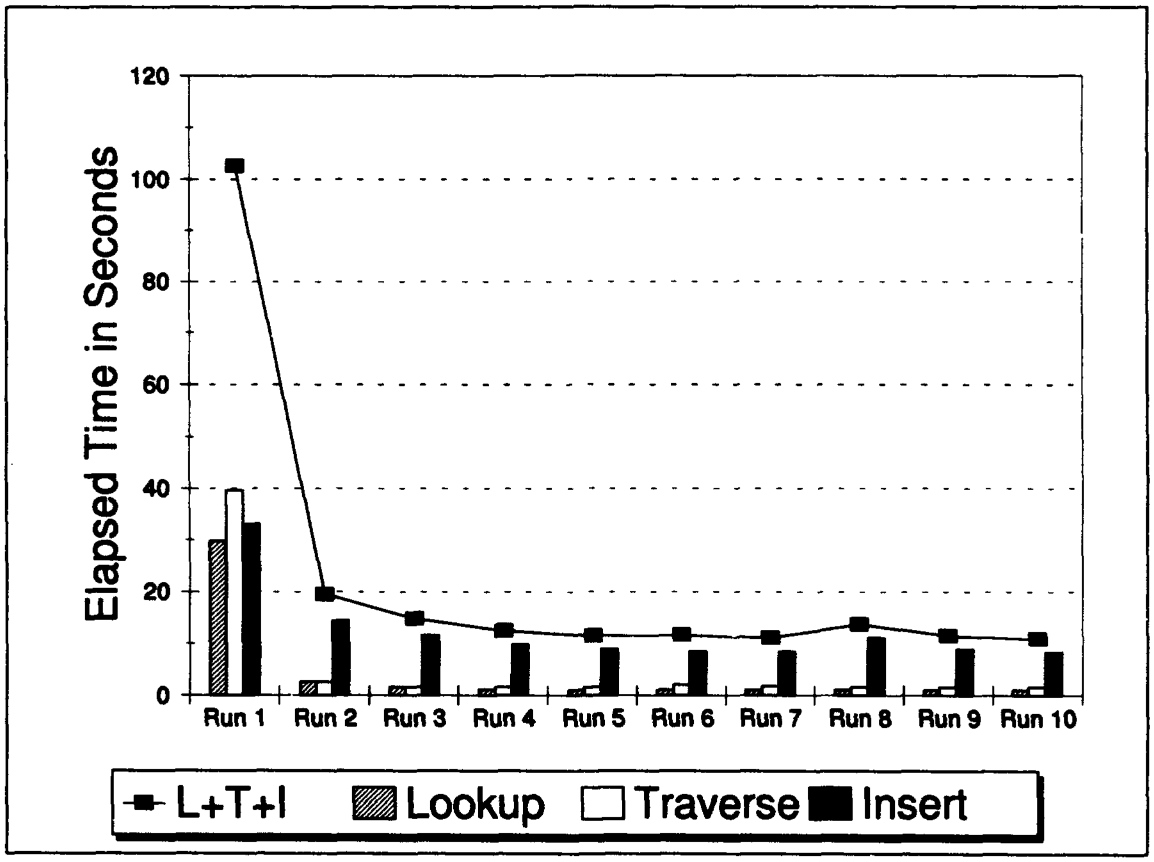

Figure 61. ObjectStore OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Remote* Database (NLOR)

Table 66. ObjectStore OO1 Summary Results for *Small Local* Database (NLOR)

	Lookup		Traversal		Insert		L+T+I	
Benchmark	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	35.091	1.172	47.544	1.888	38.622	11.011	121.257	14.071
2	32.773	1.192	46.459	1.946	38.422	12.483	117.654	15.621
3	32.323	1.530	48.047	1.953	39.724	11.167	120.094	14.650
4	33.526	1.211	50.345	1.626	38.069	12.252	121.940	15.089
5	32.530	1.453	28.688	1.546	40.675	12.655	101.893	15.654
Average:	33.249	1.312	44.217	1.792	39.102	11.914	116.568	15.017
Sample STD:	1.126	0.167	8.796	0.192	1.075	0.768	8.364	0.672

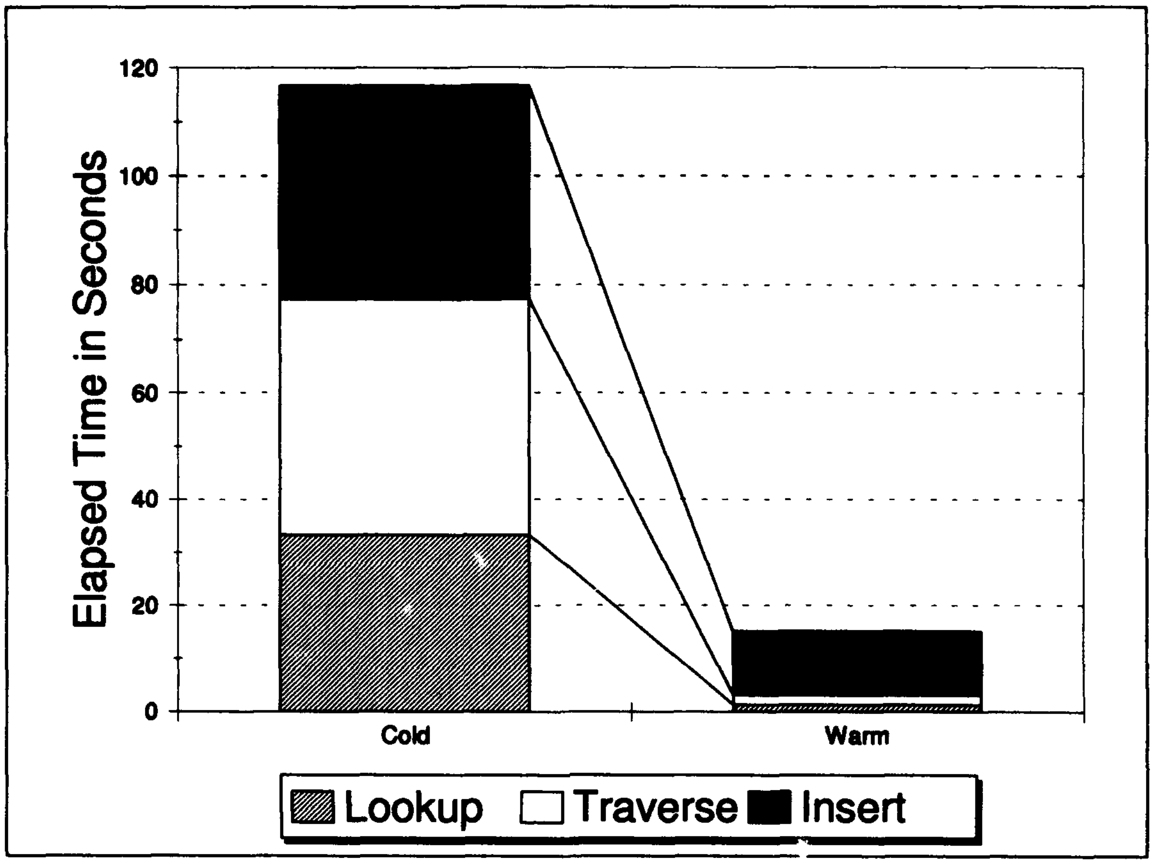

Figure 62. ObjectStore OO1 Average Benchmark Results for *Small Local* Database (NLOR)

Table 67. ObjectStore OO1 Normalized Reverse Traversal Results for *Small Local* Database (NLOR)

Benchmark	Cold	Warm
1	19.324	47.308
2	227.614	35.472
3	121.962	2.120
4	45.424	1.661
5	106.063	2.388
Average:	104.077	17.790
Sample STD:	80.945	21.948

Table 68. ObjectStore OO1 Raw Benchmark Results for *Small Local* Database (NLOR)

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	35.091	2.393	1.164	1.009	0.980	0.997	0.996	1.004	0.995	1.009
	Traversal	47.544	2.030	1.566	1.580	1.502	1.552	1.505	1.513	4.193	1.555
	Insert	38.622	13.906	11.152	10.078	12.940	9.204	13.831	9.761	9.428	8.796
	R. Traversal	51.139	1.633	1.525	1.523	1.503	1.605	1.632	1.611	0.125	1.480
2	Lookup	32.773	2.509	1.271	0.988	0.989	0.985	1.010	0.989	1.005	0.986
	Traversal	46.459	2.174	1.643	1.536	1.540	1.542	4.390	1.564	1.565	1.556
	Insert	38.422	16.824	16.447	12.145	11.427	9.562	13.256	9.760	13.461	9.461
	R. Traversal	40.318	10.270	0.092	1.400	1.717	1.684	1.616	1.620	1.696	1.651
3	Lookup	32.323	5.718	1.137	0.994	0.988	0.984	0.995	0.990	0.982	0.983
	Traversal	48.047	2.129	1.561	1.558	1.551	4.607	1.553	1.525	1.571	1.522
	Insert	39.724	15.396	11.532	9.831	13.529	9.545	13.561	9.060	8.896	9.149
	R. Traversal	51.834	2.847	1.655	1.687	1.684	1.644	1.552	1.614	1.735	1.403
4	Lookup	33.526	2.733	1.149	1.056	0.983	1.004	0.991	0.999	0.993	0.991
	Traversal	50.345	2.289	1.582	1.558	1.516	1.556	1.538	1.586	1.518	1.488
	Insert	38.069	18.679	14.315	15.512	10.945	13.487	10.111	9.029	9.127	9.062
	R. Traversal	51.005	1.753	1.714	1.693	1.622	1.719	1.622	1.627	1.564	4.348
5	Lookup	32.530	2.340	1.161	1.138	0.995	1.002	1.018	1.013	3.413	0.995
	Traversal	28.688	1.738	1.525	1.532	1.512	1.530	1.526	1.524	1.506	1.521
	Insert	40.675	21.021	12.712	14.398	10.396	10.866	9.758	11.945	9.744	13.057
	R. Traversal	47.567	6.280	0.965	1.667	1.498	1.463	1.743	1.727	1.615	1.681

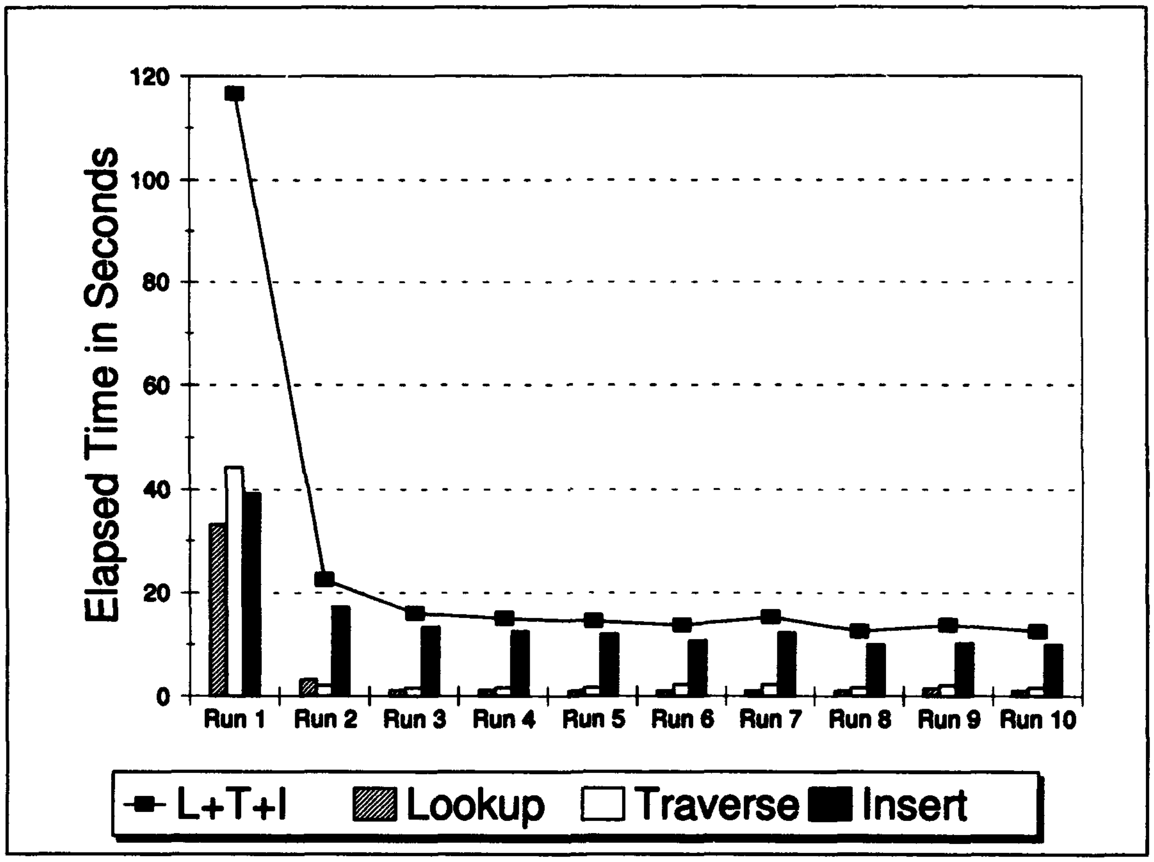

Figure 63. ObjectStore OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Small Local* Database (NLOR)

Table 69. ObjectStore OO1 Summary Results for *Large Remote* Database

Benchmark	Lookup		Traversal		Insert		L+T+I	
	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	133.030	63.529	166.470	116.962	98.235	57.737	397.735	238.228
2	115.217	63.876	219.191	114.019	127.410	61.409	461.818	239.304
3	113.467	57.615	199.172	118.560	97.724	61.939	410.363	238.114
4	113.047	60.975	211.815	118.342	98.511	62.557	423.373	241.874
5	134.295	74.098	176.842	122.197	98.752	63.007	409.889	259.302
Average:	121.811	64.019	194.698	118.016	104.126	61.330	420.636	243.364
Sample STD:	10.858	6.167	22.520	2.957	13.022	2.098	24.743	9.037

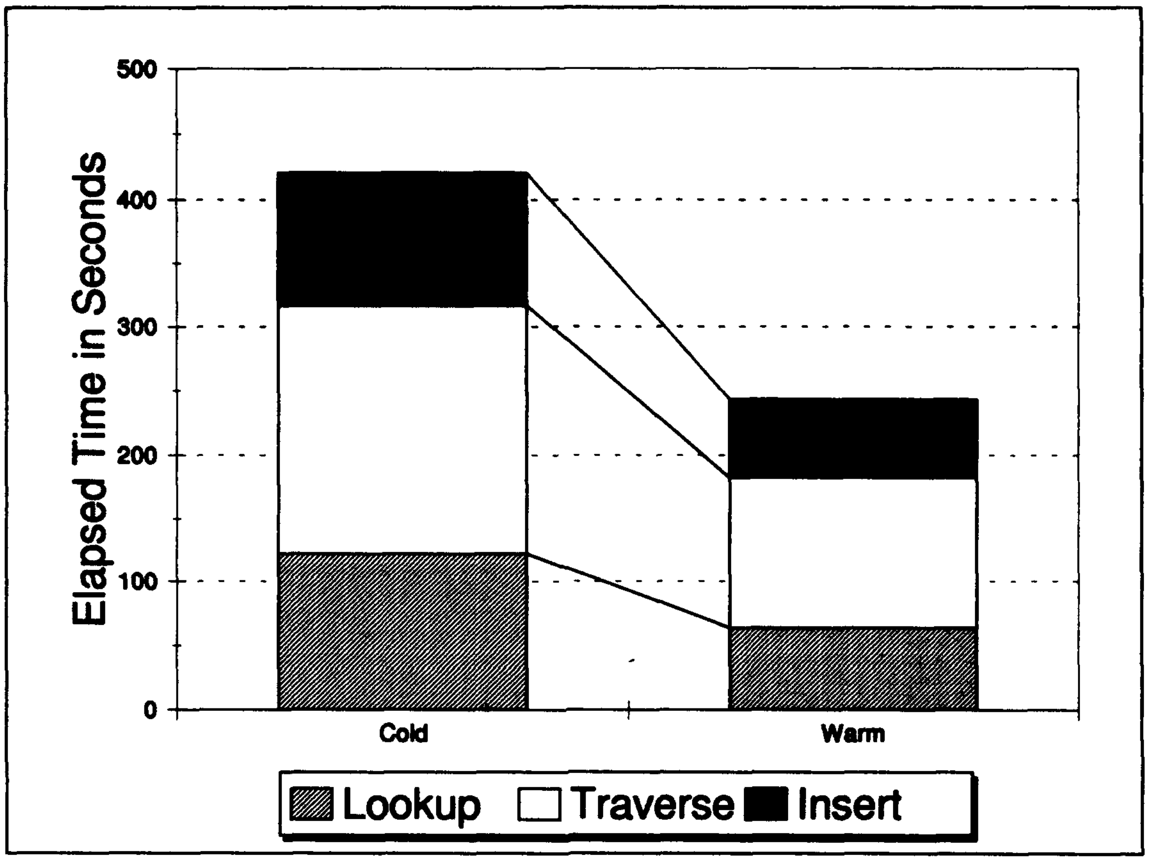

Figure 64. ObjectStore OO1 Average Benchmark Results for *Large Remote* Database

Table 70. ObjectStore OO1 Normalized Reverse Traversal Results for *Large Remote* Database

Benchmark	Cold	Warm
1	236.755	187.308
2	277.469	469.744
3	220.835	195.891
4	8785.680	584.968
5	342.571	204.768
Average:	1972.662	328.536
Sample STD:	3808.883	186.115

Table 71. ObjectStore OO1 Raw Benchmark Results for *Large Remote* Database

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	133.030	88.507	65.207	62.236	53.802	60.255	55.136	63.080	64.086	59.455
	Traversal	166.470	150.306	108.219	127.313	113.022	95.674	107.236	104.902	132.390	113.596
	Insert	98.235	54.531	52.037	64.335	57.882	56.116	58.597	60.028	59.145	56.962
	R. Traversal	311.752	242.870	230.778	102.856	227.474	188.182	58.921	171.092	71.877	280.807
2	Lookup	115.217	60.712	62.156	70.734	54.552	61.566	54.034	71.237	69.621	70.272
	Traversal	219.191	131.767	109.717	122.110	118.880	99.703	110.496	110.956	115.466	107.077
	Insert	127.410	67.813	60.207	62.266	67.362	56.639	56.977	60.518	58.065	62.838
	R. Traversal	629.211	0.760	337.673	52.986	0.498	25.359	219.435	306.428	74.413	474.474
3	Lookup	113.467	61.217	52.514	59.268	52.207	59.677	54.251	60.603	55.682	63.113
	Traversal	199.172	152.171	118.421	125.950	123.678	104.219	109.911	111.342	110.313	111.039
	Insert	97.724	60.883	61.078	65.288	57.813	59.307	59.839	64.271	66.294	62.678
	R. Traversal	377.372	64.442	445.243	337.698	137.476	208.464	35.405	301.526	29.511	116.489
4	Lookup	113.047	62.989	56.071	62.074	63.176	55.811	63.129	62.575	58.714	64.235
	Traversal	211.815	111.149	133.310	112.906	118.542	118.601	116.751	125.835	114.235	113.752
	Insert	98.511	64.428	57.596	67.207	61.881	59.506	58.692	67.362	63.950	62.391
	R. Traversal	2.679	0.562	1.035	147.614	139.358	214.363	42.627	97.060	145.012	74.411
5	Lookup	134.295	108.627	98.508	66.045	55.891	84.310	57.725	65.938	67.749	62.090
	Traversal	176.842	144.329	137.031	138.406	119.691	107.443	117.294	112.898	120.905	101.777
	Insert	98.752	67.806	63.573	65.616	65.765	63.357	69.824	55.447	62.798	52.876
	R. Traversal	118.960	311.815	214.997	135.674	383.921	147.019	618.172	156.057	32.253	99.512

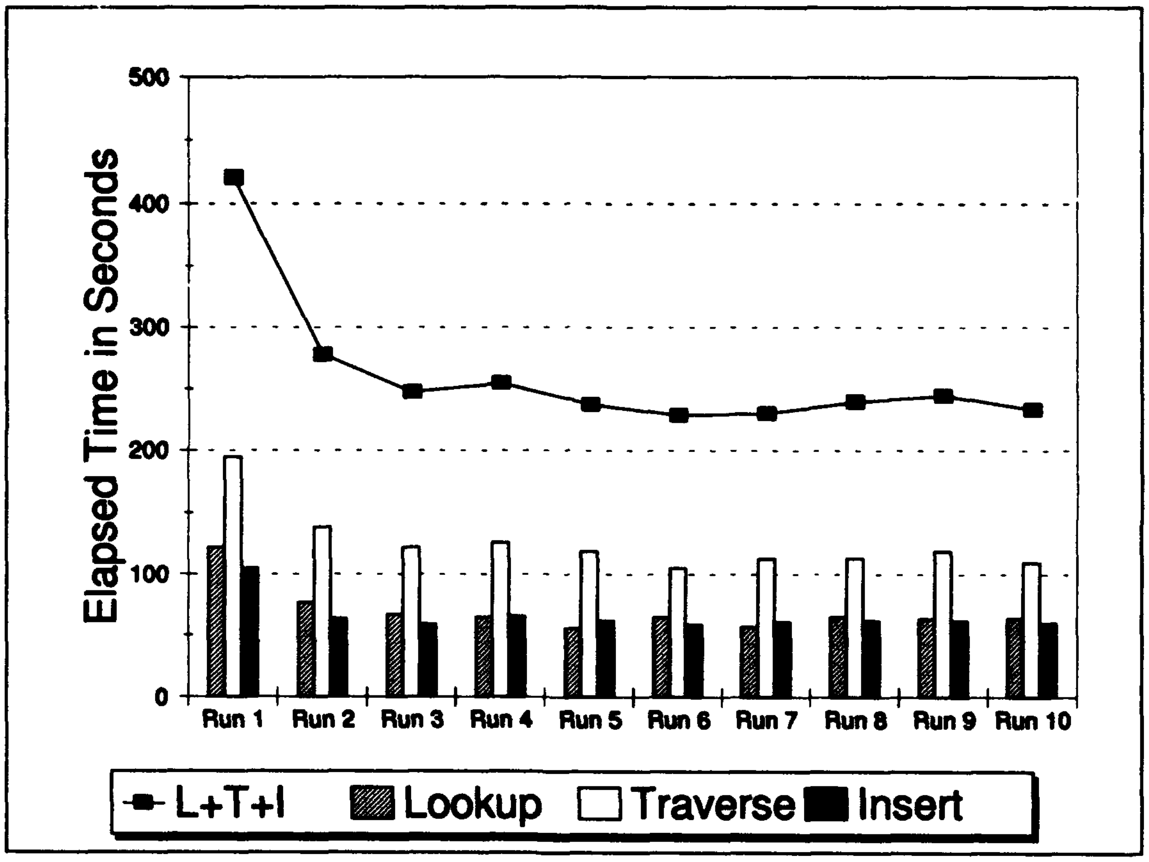

Figure 65. ObjectStore OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Large Remote* Database

Table 72. ObjectStore OO1 Summary Results for *Large Local* Database

	Lookup		Traversal		Insert		L+T+I	
Benchmark	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
1	106.016	54.702	164.276	151.364	80.594	44.599	350.886	250.665
2	107.407	53.033	191.239	153.353	68.920	45.286	367.566	251.672
3	128.207	67.027	169.382	161.498	66.981	43.856	364.570	272.381
4	84.170	55.513	219.481	152.014	69.924	45.504	373.575	253.031
5	101.517	53.492	172.987	161.155	66.246	50.279	340.750	264.926
Average:	105.463	56.753	183.473	155.877	70.533	45.905	359.469	258.535
Sample STD:	15.735	5.826	22.550	5.028	5.814	2.529	13.366	9.642

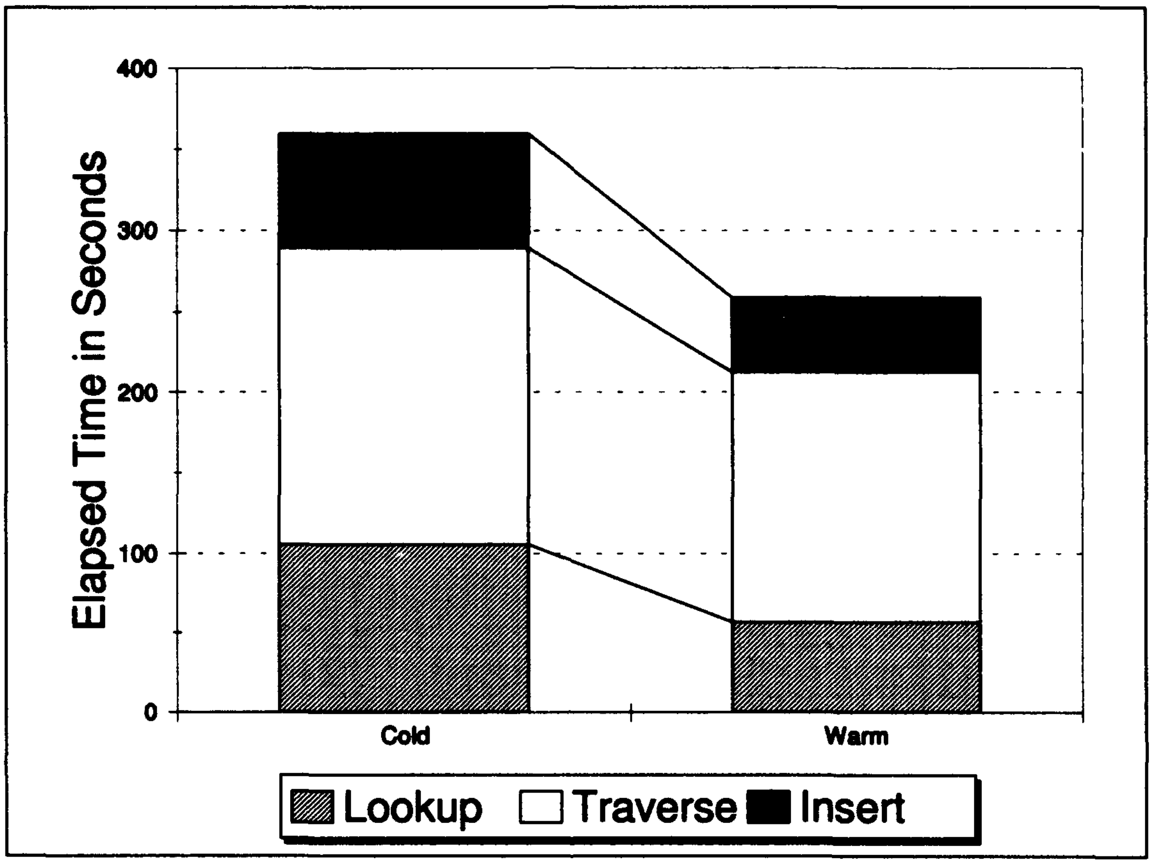

Figure 66. ObjectStore OO1 Average Benchmark Results for *Large Local* Database

Table 73. ObjectStore OO1 Normalized Reverse Traversal Results for *Large Local* Database

Benchmark	Cold	Warm
1	285.299	277.239
2	294.898	638.452
3	291.708	291.331
4	17787.702	679.028
5	329.497	291.428
Average:	3797.821	435.496
Sample STD:	7820.600	204.379

Table 74. ObjectStore OO1 Raw Benchmark Results for *Large Local* Database

Benchmark	Measure	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	Lookup	106.016	47.458	47.009	49.047	58.158	51.382	62.707	55.619	64.601	56.339
	Traversal	164.276	158.620	144.393	159.795	161.003	131.889	149.073	146.267	162.550	148.683
	Insert	80.594	49.202	45.420	46.128	45.485	42.797	44.568	39.547	41.598	46.649
	R. Traversal	375.672	310.940	329.559	145.974	332.608	279.804	100.215	258.244	111.117	434.572
2	Lookup	107.407	48.901	47.897	56.198	48.948	50.189	59.685	51.931	61.485	52.068
	Traversal	191.239	154.732	143.225	143.536	164.353	142.576	152.266	156.966	170.735	151.791
	Insert	68.920	34.413	41.213	48.747	49.147	51.514	44.797	43.346	48.291	46.104
	R. Traversal	668.736	0.979	494.859	71.281	0.653	52.874	314.885	449.170	114.035	577.800
3	Lookup	128.207	113.317	90.744	62.961	49.312	51.370	61.614	53.171	63.230	57.526
	Traversal	169.382	152.080	168.728	172.035	168.252	154.706	155.290	158.206	159.105	165.084
	Insert	66.981	50.949	41.358	39.421	43.277	42.826	44.595	45.417	44.011	42.848
	R. Traversal	498.483	91.764	632.747	499.204	204.500	315.226	57.640	452.056	41.466	179.858
4	Lookup	84.170	57.374	50.279	49.838	60.335	51.029	61.131	52.151	55.053	62.424
	Traversal	219.481	129.041	170.310	144.493	157.175	157.208	154.253	152.843	152.441	150.365
	Insert	69.924	43.850	53.361	47.643	45.494	46.395	41.007	42.783	45.374	43.632
	R. Traversal	5.423	0.696	1.128	115.911	96.222	282.062	67.726	135.136	208.418	106.575
5	Lookup	101.517	58.374	48.242	47.258	57.940	49.136	50.201	59.961	51.131	59.187
	Traversal	172.987	146.884	182.321	183.038	161.589	144.413	166.830	153.014	169.159	143.151
	Insert	66.246	51.682	43.244	42.740	47.067	45.032	58.900	55.439	53.089	55.319
	R. Traversal	114.420	438.669	300.664	185.619	528.373	203.909	909.157	239.278	46.168	144.156

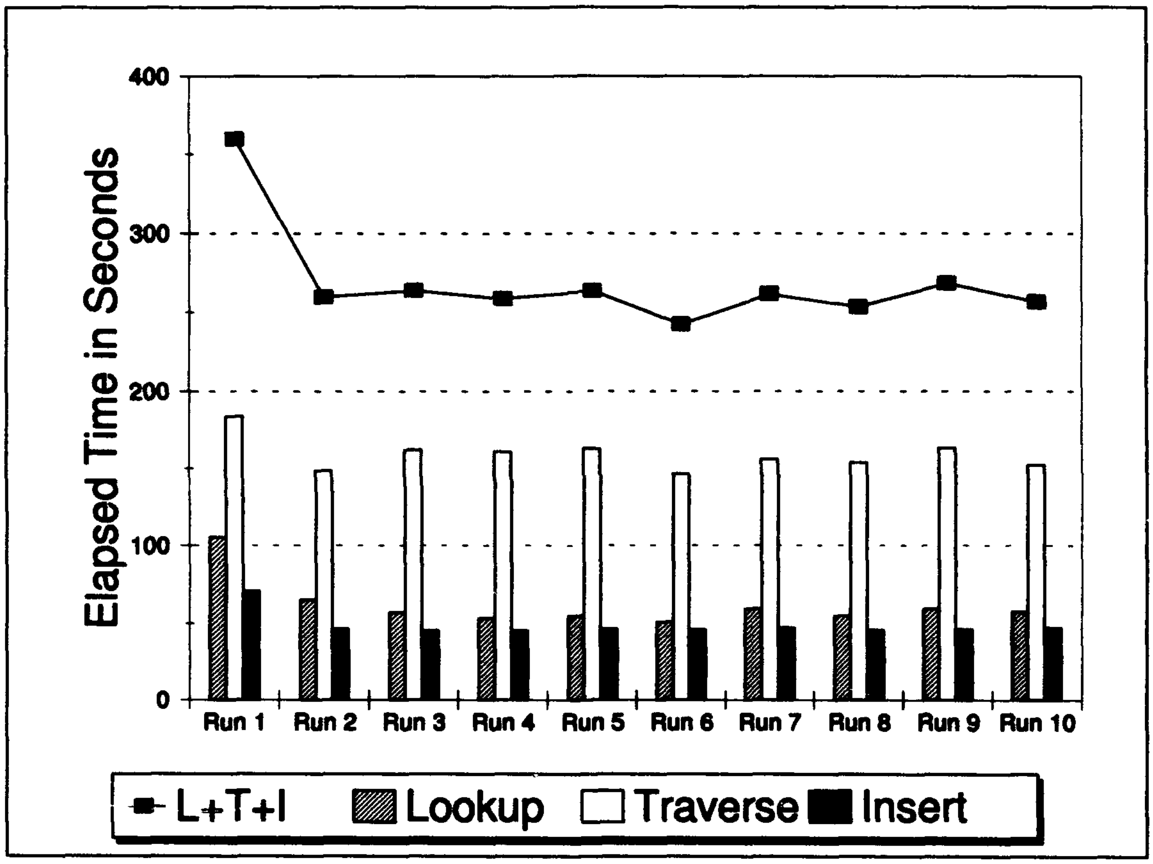

Figure 67. ObjectStore OO1 Average Individual Benchmark Measures (and Benchmark Total) Across the Ten Runs for *Large Local* Database

Appendix C. Statistical Analysis of the OO1 Benchmark Results

To determine if the differences in the OO1 benchmark results were meaningful, we ran a ***small-sample test of hypothesis for the difference between population means*** on our results. The statistical test used in this appendix was described by McClave and Benson in [26].

For the OO1 benchmark results, we were interested in determining if a difference existed between results from two different databases on the same benchmark configuration or the same database on two different benchmark configurations. For example, is the mean lookup time for the ObjectStore DBMS different than the mean lookup time for the Itasca DBMS, and if so, which is faster. The ***population*** mean μ for our benchmark results is unknown, we only know the ***sample*** mean $\bar{x}$ for our five runs. Therefore, assigning μ_1 to be the population mean of one of the results and μ_2 to be the population mean of the second, we wanted to detect a difference between μ_1 and μ_2 if and only if a difference exists. Therefore, we tested the null hypothesis shown in Equation 2,

$$H_0 : (\mu_1 - \mu_2) = 0 \tag{2}$$

against the alternative hypothesis shown in Equation 3.

$$H_a : (\mu_1 - \mu_2) \neq 0 \quad (\text{i.e., either } \mu_1 > \mu_2 \text{ or } \mu_2 > \mu_1) \tag{3}$$

To perform this test, we first calculated the ***test statistic*** using Equation 4. In Equation 4, n is the number of samples taken (which was always 5 for our work), $\bar{x}$ is the sample mean, and s is the sample standard deviation.

$$t = \frac{(\bar{x}_1 - \bar{x}_2) - 0}{\sqrt{\frac{(n_1-1)s_1^2+(n_2-1)s_2^2}{n_1+n_2-2}\left(\frac{1}{n_1} + \frac{1}{n_2}\right)}} \tag{4}$$

The test statistics calculated for the ***Small Remote*** database results are shown in Table 75. The test statistics calculated for the ***Small Local*** database results are shown in Table 76. The test statistics calculated for the ***Small Remote*** database with no locality of reference results are shown in Table 77. The test statistics calculated for the ***Small Local*** database with no locality of reference results are shown in Table 78. The test statistics calculated for the *Large Remote* database results are shown in Table 79. The test statistics calculated for the *Large Local* database results are shown in Table 80. The test statistics calculated to compare Matisse database results for test runs with no delay for version collection with test runs allowing a 200 second delay are shown in Table 81. The test statistics calculated to compare Matisse database results for a *Remote* client with a *Local* client are shown in Table 82. The test statistics calculated to compare ObjectStore database results for a *Remote* client with a *Local* client are shown in Table 83.

A calculated test statistic may be compared with the *rejection region* of our test. The rejection region is determined from Student's t distribution. The *degrees of freedom* used for our test was 8 ($n_1 + n_2 - 2 = 5 + 5 - 2 = 8$) and we choose an α of 0.05. The rejection region for our test was $t < -t_{\frac{\alpha}{2}}$ or $t > t_{\frac{\alpha}{2}}$ where $t_{\frac{\alpha}{2}}$ is based on 8 degrees of freedom. Hence, we can make one of the following conclusions:

- The first benchmark result is faster than the second benchmark result if $t < -2.306$
- The second benchmark result is faster than the first benchmark result if $t > 2.306$
- The sample evidence is insufficient to reject the null hypothesis at $\alpha = 0.05$

Our test results are statistically significant at the $\alpha = 0.05$ level of significance.

To allow the use of our statistical test, we had to assume that the population standard deviation for both samples was equal. This assumption is probably reasonable because most sample variation was due to random number variations, and system loading variations. The random number variations were the same for all samples (the same random number streams used for one group of benchmark runs, was used for them all). We controlled the system load variations (operating system, etc.) by ensuring that our benchmark was the only non-system job running on the computer during a benchmark run.

Table 75. Small-Sample Test Statistic for OO1 Benchmark *Small Remote* Database Results

DBMS		Lookup		Traversal		Insert		L+T+I	
		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
Itasca	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	277.278	213.162	347.040	251.136	134.765	129.935	759.083	594.233
	Sample STD:	81.307	20.643	100.550	33.854	1.987	2.311	175.232	49.393
Matisse	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	125.982	68.448	53.111	35.387	64.076	49.507	243.169	153.382
	Sample STD:	35.057	16.721	9.689	0.851	2.496	1.192	36.720	16.153
	t =	3.821	12.177	6.506	14.246	49.545	69.162	6.443	18.969
Matisse	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	125.982	68.448	53.111	35.387	64.076	49.507	243.169	153.382
	Sample STD:	35.057	16.721	9.689	0.851	2.496	1.192	36.720	16.153
ObjectStore	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	28.191	1.239	26.322	1.734	16.285	3.184	70.798	6.157
	Sample STD:	2.701	0.111	3.173	0.169	1.012	0.418	5.722	0.476
	t =	6.219	8.993	5.875	86.732	39.677	82.001	10.371	20.372
Itasca	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	277.278	213.162	347.040	251.136	134.765	129.935	759.083	594.233
	Sample STD:	81.307	20.643	100.550	33.854	1.987	2.311	175.232	49.393
ObjectStore	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	28.191	1.239	26.322	1.734	16.285	3.184	70.798	6.157
	Sample STD:	2.701	0.111	3.173	0.169	1.012	0.418	5.722	0.476
	t =	6.847	22.955	7.129	16.473	118.809	120.683	8.778	26.622

Table 76. Small-Sample Test Statistic for OO1 Benchmark *Small Local* Database Results

DBMS		Lookup		Traversal		Insert		L+T+I	
		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
Itasca	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	251.064	200.107	248.967	211.769	130.855	120.430	630.886	532.306
	Sample STD:	28.810	11.376	6.828	19.502	3.690	1.205	29.944	31.162
Matisse	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	91.652	36.058	70.499	41.030	64.279	49.899	226.431	126.987
	Sample STD:	18.883	7.119	16.389	6.850	2.153	1.603	30.136	10.240
	t =	10.348	27.334	22.447	18.470	34.846	78.644	21.288	27.631
Matisse	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	91.652	36.058	70.499	41.030	64.279	49.899	226.431	126.987
	Sample STD:	18.883	7.119	16.389	6.850	2.153	1.603	30.136	10.240
ObjectStore	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	28.859	1.274	29.029	2.204	17.791	3.264	75.680	6.741
	Sample STD:	1.011	0.122	5.664	0.405	1.427	0.639	5.280	0.621
	t =	7.425	10.924	5.348	12.652	40.244	60.428	11.018	26.209
Itasca	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	251.064	200.107	248.967	211.769	130.855	120.430	630.886	532.306
	Sample STD:	28.810	11.376	6.828	19.502	3.690	1.205	29.944	31.162
ObjectStore	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	28.859	1.274	29.029	2.204	17.791	3.264	75.680	6.741
	Sample STD:	1.011	0.122	5.664	0.405	1.427	0.639	5.280	0.621
	t =	17.236	39.080	55.436	24.023	63.903	192.083	40.830	37.705

Table 77. Small-Sample Test Statistic for OO1 Benchmark *Small Remote* Database Results (NLOR)

		Lookup		Traversal		Insert		L+T+I	
DBMS		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
Itasca	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	252.481	203.955	380.217	301.063	206.002	160.953	838.700	665.972
	Sample STD:	25.189	14.212	57.378	50.142	43.559	6.399	84.269	61.445
Matisse	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	146.186	78.955	120.816	63.672	88.340	61.556	355.342	204.182
	Sample STD:	43.697	25.572	6.326	3.279	5.082	1.288	39.033	25.181
	t =	4.712	9.554	10.048	10.564	5.999	34.050	11.638	15.550
Matisse	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	146.186	78.955	120.816	63.672	88.340	61.556	355.342	204.182
	Sample STD:	43.697	25.572	6.326	3.279	5.082	1.288	39.033	25.181
ObjectStore	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	29.765	1.250	39.574	1.766	33.045	9.919	102.384	12.936
	Sample STD:	4.506	0.091	6.787	0.122	4.970	1.196	4.982	1.102
	t =	5.926	6.795	19.580	42.187	17.394	65.692	14.374	16.966
Itasca	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	252.481	203.955	380.217	301.063	206.002	160.953	838.700	665.972
	Sample STD:	25.189	14.212	57.378	50.142	43.559	6.399	84.269	61.445
ObjectStore	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	29.765	1.250	39.574	1.766	33.045	9.919	102.384	12.936
	Sample STD:	4.506	0.091	6.787	0.122	4.970	1.196	4.982	1.102
	t =	19.462	31.892	13.183	13.347	8.821	51.879	19.504	23.761

Table 78. Small-Sample Test Statistic for OO1 Benchmark *Small Local* Database Results (NLOR)

		Lookup		Traversal		Insert		L+T+I	
DBMS		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
Itasca	N:	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	Average:	234.236	197.707	342.590	270.023	188.591	150.635	765.417	618.365
	Sample STD:	*none*	*none*	*none*	*none*	*none*	*none*	*none*	*none*
Matisse	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	107.941	31.779	145.648	58.183	98.353	61.876	351.942	151.839
	Sample STD:	38.749	3.251	45.573	4.582	17.174	2.728	79.553	8.598
	t =								
Matisse	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	107.941	31.779	145.648	58.183	98.353	61.876	351.942	151.839
	Sample STD:	38.749	3.251	45.573	4.582	17.174	2.728	79.553	8.598
ObjectStore	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	33.249	1.312	44.217	1.792	39.102	11.914	116.568	15.017
	Sample STD:	1.126	0.167	8.796	0.192	1.075	0.768	8.364	0.672
	t =	4.308	20.928	4.887	27.495	7.699	39.420	6.580	35.475
Itasca	N:	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	Average:	234.236	197.707	342.590	270.023	188.591	150.635	765.417	618.365
	Sample STD:	*none*	*none*	*none*	*none*	*none*	*none*	*none*	*none*
ObjectStore	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	33.249	1.312	44.217	1.792	39.102	11.914	116.568	15.017
	Sample STD:	1.126	0.167	8.796	0.192	1.075	0.768	8.364	0.672
	t =								

Table 79. Small-Sample Test Statistic for OO1 Benchmark *Large Remote* Database Results

DBMS		Lookup Cold	Lookup Warm	Traversal Cold	Traversal Warm	Insert Cold	Insert Warm	L+T+I Cold	L+T+I Warm
Matisse	N:	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0
	Average:	245.811	216.350	231.703	214.405	119.756	110.767	577.456	535.937
	Sample STD:	38.353	28.555	47.183	12.560	4.380	3.350	35.431	31.677
ObjectStore	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	121.811	64.019	194.698	118.016	104.126	61.330	420.636	243.364
	Sample STD:	10.858	6.167	22.520	2.957	13.022	2.098	24.743	9.037
	t =	6.956	11.660	1.583	16.704	2.544	27.967	8.114	19.866

Table 80. Small-Sample Test Statistic for OO1 Benchmark *Large Local* Database Results

DBMS		Lookup		Traversal		Insert		L+T+I	
		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
Matisse	N:	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0
	Average:	228.080	215.053	229.644	228.602	106.308	112.559	546.682	552.772
	Sample STD:	22.379	22.185	41.520	12.606	3.171	1.433	21.878	24.174
ObjectStore	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	105.463	56.753	183.473	155.877	70.533	45.905	359.469	258.535
	Sample STD:	15.735	5.826	22.550	5.028	5.814	2.529	13.366	9.642
	t =	10.022	15.432	2.185	11.982	12.079	51.274	16.328	25.280

Table 81. Small-Sample Test Statistic for OO1 Benchmark Matisse Results (No Delay/200 Second Delay)

		Lookup		Traversal		Insert		L+T+I	
DBMS		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
Matisse	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
(*Small Local—No Delay*)	Average:	91.652	36.058	70.499	41.030	64.279	49.899	226.431	126.987
	Sample STD:	18.883	7.119	16.389	6.850	2.153	1.603	30.136	10.240
Matisse	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
(*Small Local—Delay*)	Average:	112.970	32.179	75.423	38.728	66.496	48.418	254.889	119.325
	Sample STD:	38.827	2.700	26.108	9.899	11.648	3.592	74.101	13.564
	t =	-1.104	1.139	-0.357	0.428	-0.419	0.842	-0.795	1.008
Matisse	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
(*Small Remote NLOR—No Delay*)	Average:	146.186	78.955	120.816	63.672	88.340	61.556	355.342	204.182
	Sample STD:	43.697	25.572	6.326	3.279	5.082	1.288	39.033	25.181
Matisse	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
(*Small Remote NLOR—Delay*)	Average:	136.758	67.555	115.521	63.166	90.190	62.389	342.470	193.111
	Sample STD:	46.450	18.914	14.383	0.690	4.193	1.474	47.241	19.993
	t =	0.331	0.801	0.754	0.338	-0.628	-0.952	0.470	0.770

Table 82. Small-Sample Test Statistic for OO1 Benchmark Matisse *Local/Remote* Database Results

DBMS		Lookup		Traversal		Insert		L+T+I	
		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
Matisse	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
(*Small Remote*)	Average:	125.982	68.448	53.111	35.387	64.076	49.507	243.169	153.382
	Sample STD:	35.057	16.721	9.689	0.851	2.496	1.192	36.720	16.153
Matisse	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
(*Small Local*)	Average:	91.652	36.058	70.499	41.030	64.279	49.899	226.431	126.987
	Sample STD:	18.883	7.119	16.389	6.850	2.153	1.603	30.136	10.240
	t =	1.928	3.990	-2.042	-1.828	-0.138	-0.439	0.778	3.086
Matisse	N:	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0
(*Large Remote*)	Average:	245.811	216.350	231.703	214.405	119.756	110.767	577.456	535.937
	Sample STD:	38.353	28.555	47.183	12.560	4.380	3.350	35.431	31.677
Matisse	N:	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0
(*Large Local*)	Average:	228.080	215.053	229.644	228.602	106.308	112.559	546.682	552.772
	Sample STD:	22.379	22.185	41.520	12.606	3.171	1.433	21.878	24.174
	t =	0.893	0.080	0.073	-1.784	5.561	-1.100	1.653	-0.945

Table 83. Small-Sample Test Statistic for OO1 Benchmark ObjectStore *Local/Remote* Database Results

		Lookup		Traversal		Insert		L+T+I	
DBMS		Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
ObjectStore	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
(*Small Remote*)	Average:	28.191	1.239	26.322	1.734	16.285	3.184	70.798	6.157
	Sample STD:	2.701	0.111	3.173	0.169	1.012	0.418	5.722	0.476
ObjectStore	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
(*Small Local*)	Average:	28.859	1.274	29.029	2.204	17.791	3.264	75.680	6.741
	Sample STD:	1.011	0.122	5.664	0.405	1.427	0.639	5.280	0.621
	t =	-0.518	-0.474	-0.932	-2.395	-1.925	-0.234	-1.402	-1.669
ObjectStore	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
(*Large Remote*)	Average:	121.811	64.019	194.698	118.016	104.126	61.330	420.636	243.364
	Sample STD:	10.858	6.167	22.520	2.957	13.022	2.098	24.743	9.037
ObjectStore	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
(*Large Local*)	Average:	105.463	56.753	183.473	155.877	70.533	45.905	359.469	258.535
	Sample STD:	15.735	5.826	22.550	5.028	5.814	2.529	13.366	9.642
	t =	1.912	1.915	0.788	-14.514	5.276	10.497	4.864	-2.567

Appendix D. Detailed Simulation Benchmark Results

This appendix contains, in detail, the results obtained from our runs of the simulation benchmark. Benchmark results are included for the ObjectStore DBMS version of the benchmark and the non-persistent version of the benchmark.

For each different benchmark configuration, five complete runs of the benchmark were made. Five runs were made rather than one to try to obtain a better picture of typical performance, not just a single snapshot. This appendix contains both raw and summary results for all the benchmark configurations. Benchmark results are reported using two tables and one chart. These are described below in the order which they appear for each benchmark configuration.

- *Summary Results Table*: This table reports the times for all five runs of the simulation benchmark. The average and standard deviation of the five runs is given at the bottom of the table. All the times reported in this table are in seconds.
- *Throughput Results Table*: This table reports the observed actual time ratios during the throughput measure. The time ratio of the simulation is defined as the ratio of wall clock time to simulation time. The reported value for the throughput measure is the 90th percentile of the samples, but this table also reports the geometric mean.
- *Hour Run Results Chart*: This chart provides a clear picture of the effect time slice has on the hour run results. The time slice defines the transaction size for the object-oriented DBMS. As the time slice increases, less transactions are run during the simulated hour. It is important to note that the axis scales of this chart are different for each benchmark configuration, so care must be taken when using this chart for comparisons.

D.0.1 Simulation Benchmark Results for the ObjectStore DBMS. This section reports our results for the ObjectStore DBMS version of the simulation benchmark. Results for the following database configurations are provided:

- ObjectStore *Small Remote* Database — The results for this benchmark configuration are provided in Tables 84 and 85 and in Figure 68.
- ObjectStore *Small Local* Database — The results for this benchmark configuration are provided in Tables 86 and 87 and in Figure 69.

Table 84. ObjectStore Simulation Benchmark Summary Results for *Small Remote* Database

	Model	Scenario	Hour Run				Version	Map	Report
Benchmark	Creation	Creation	TS = 60	TS = 600	TS = 1800	TS = 3600	Creation	Creation	Creation
1	0.793	4.475	583.684	485.266	476.580	473.755	21.018	10.114	2.557
2	0.648	4.070	570.221	484.782	476.603	476.139	20.679	9.668	2.342
3	0.605	4.097	574.349	485.691	476.411	475.131	20.654	9.614	3.192
4	0.602	4.230	576.772	485.882	476.526	474.730	20.721	9.621	2.493
5	0.621	4.047	579.445	485.084	475.969	475.150	20.898	10.008	2.479
Average:	0.654	4.184	576.894	485.341	476.418	474.981	20.794	9.805	2.613
Sample STD:	0.080	0.178	5.090	0.447	0.262	0.860	0.157	0.238	0.333

Table 85. ObjectStore Simulation Benchmark Throughput Results for *Small Remote* Database

Throughput Samples				
0.123	0.139	0.143	0.148	0.148
0.150	0.151	0.151	0.153	0.155
0.155	0.155	0.156	0.157	0.157
0.157	0.158	0.159	0.159	0.161
			Geometric Mean:	0.151
			90th Percentile:	0.159

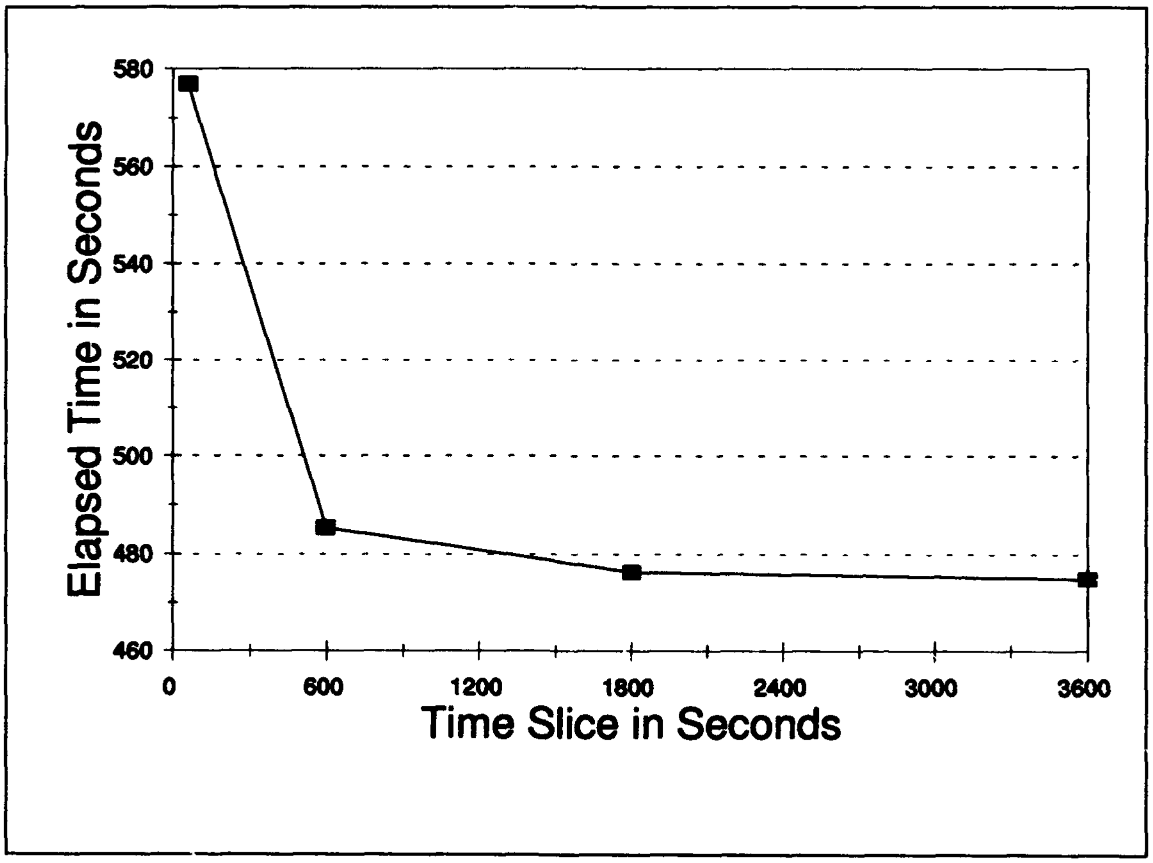

Figure 68. ObjectStore Simulation Benchmark Hour Run Results by Time Slice Setting for *Small Remote* Database

Table 86. ObjectStore Simulation Benchmark Summary Results for *Small Local* Database

Benchmark	Model Creation	Scenario Creation	Hour Run TS = 60	TS = 600	TS = 1800	TS = 3600	Version Creation	Map Creation	Report Creation
1	0.811	5.050	581.296	488.609	482.269	479.595	21.543	10.042	2.335
2	0.593	4.871	580.017	493.035	488.193	476.604	20.835	10.489	2.500
3	0.591	3.885	580.679	494.207	482.962	476.379	22.068	10.696	2.979
4	0.599	3.783	580.030	490.099	477.448	477.573	21.216	9.955	2.719
5	0.570	3.834	579.446	488.947	482.949	475.916	20.176	10.031	4.464
Average:	0.633	4.285	580.294	490.979	482.764	477.213	21.168	10.243	2.999
Sample STD:	0.100	0.621	0.710	2.508	3.810	1.462	0.716	0.329	0.854

Table 87. ObjectStore Simulation Benchmark Throughput Results for *Small Local* Database

Throughput Samples				
0.140	0.145	0.146	0.146	0.151
0.152	0.152	0.152	0.154	0.155
0.155	0.159	0.160	0.163	0.163
0.163	0.170	0.179	0.191	0.202
			Geometric Mean:	0.159
			90th Percentile:	0.180

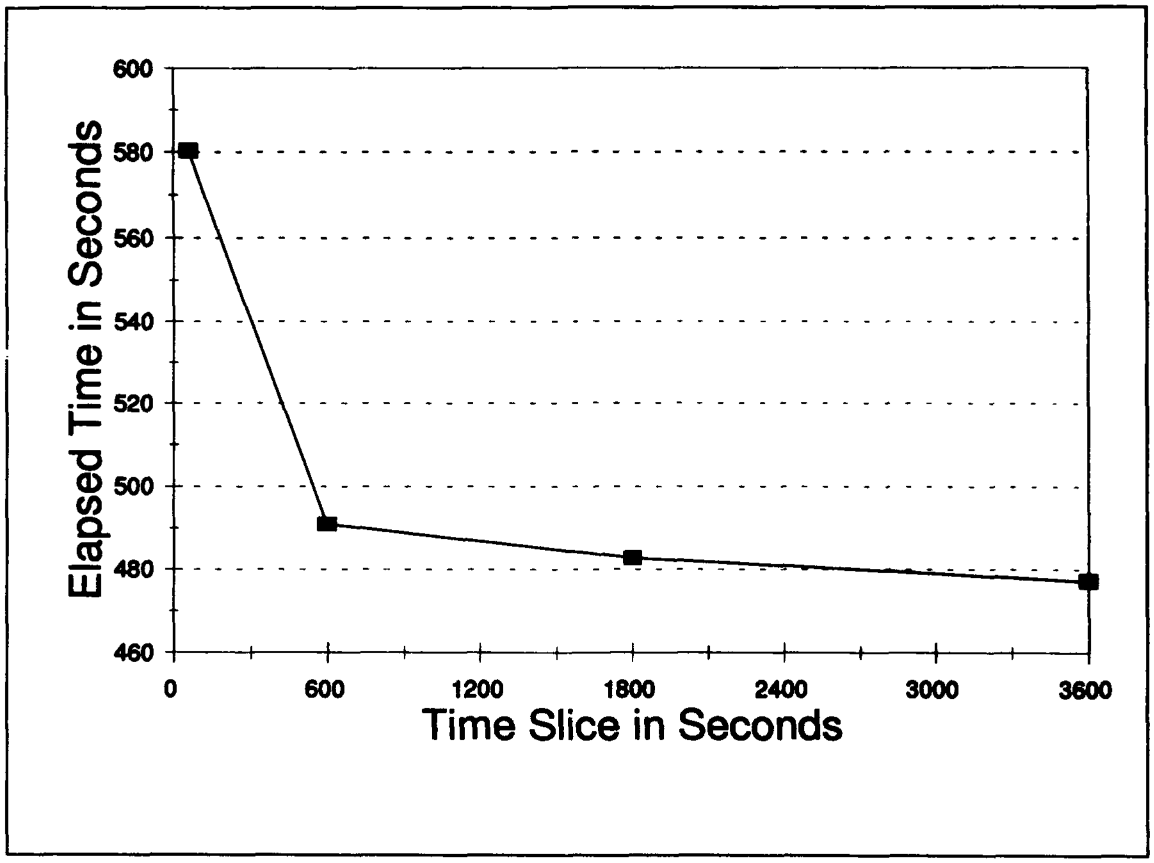

Figure 69. ObjectStore Simulation Benchmark Hour Run Results by Time Slice Setting for *Small Local* Database

D.0.2 Non-Persistent Simulation Benchmark Results. This section reports our results for the non-persistent version of the simulation benchmark. Results for the following database configurations are provided:

- Non-Persistent *Small* Database — The results for this benchmark configuration are provided in Tables 88 and 89 and in Figure 70.

Table 88. Non-Persistent Simulation Benchmark Summary Results for *Small* Database

Benchmark	Model Creation	Scenario Creation	Hour Run TS = 60	TS = 600	TS = 1800	TS = 3600	Version Creation	Map Creation	Report Creation
1	0.012	1.640	394.226	388.802	387.839	387.268	14.157	10.468	2.267
2	0.012	1.699	393.765	389.628	388.605	387.018	14.121	11.200	2.291
3	0.013	1.559	393.307	389.447	387.516	387.223	14.218	10.643	2.609
4	0.000	1.545	392.974	389.979	388.771	388.029	14.086	10.546	2.407
5	0.000	1.571	389.417	390.025	388.811	388.888	14.108	10.731	2.108
Average:	0.007	1.603	392.738	389.576	388.308	387.685	14.138	10.718	2.336
Sample STD:	0.007	0.065	1.916	0.496	0.592	0.774	0.052	0.287	0.186

Table 89. Non-Persistent Simulation Benchmark Throughput Results for *Small* Database

Throughput Samples				
0.102	0.105	0.105	0.105	0.105
0.107	0.107	0.108	0.110	0.110
0.111	0.112	0.112	0.113	0.114
0.115	0.115	0.115	0.116	0.163
			Geometric Mean:	0.112
			90th Percentile:	0.115

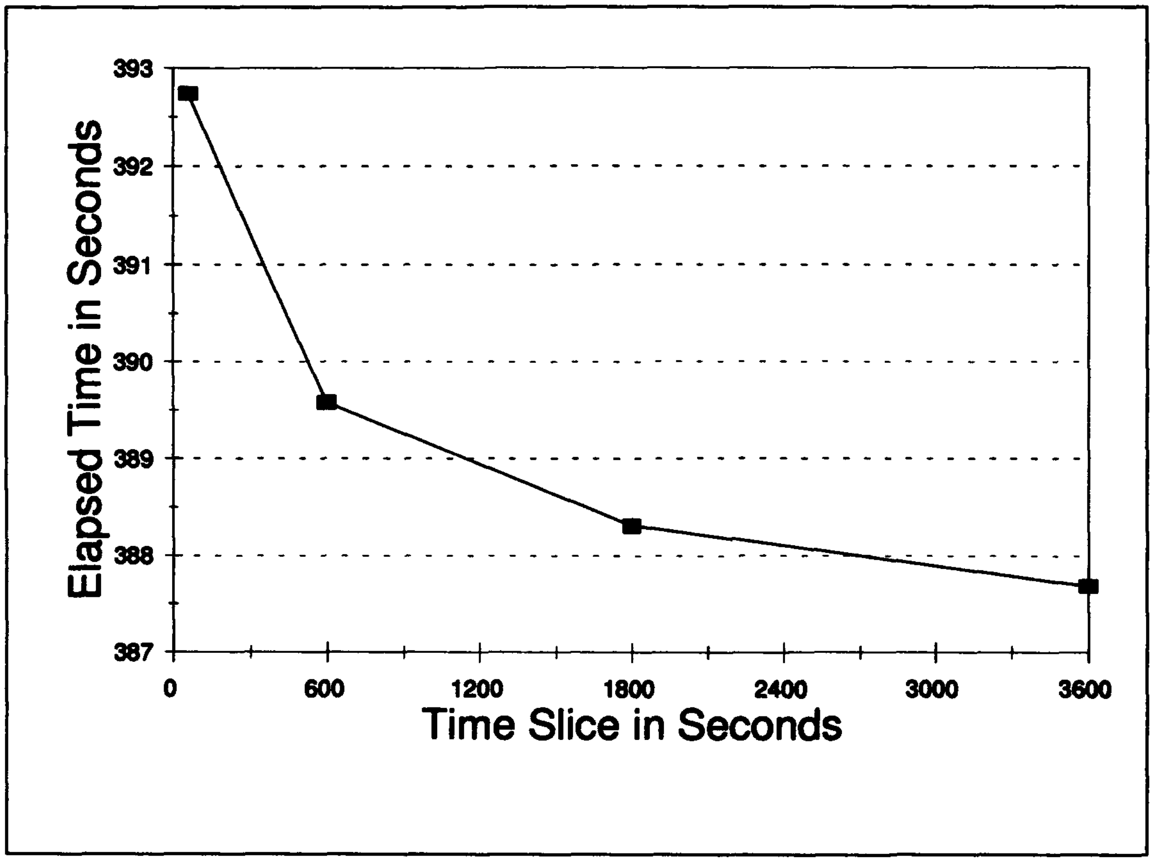

Figure 70. Non-Persistent Simulation Benchmark Hour Run Results by Time Slice Setting for *Small* Database

Appendix E. Statistical Analysis of the Simulation Benchmark Results

To determine if the differences in the simulation benchmark results were meaningful we ran a ***small-sample test of hypothesis for the difference between population means*** on our results. The statistical test used in this appendix was described by McClave and Benson in [26]. This test was also used in Appendix C.

For the simulation benchmark results we were interested in determining if a difference existed between results from two different benchmark configurations.

The calculated test statistic is compared with the *rejection region* of our test. The test statistics calculated for the *Small* database results are shown in Table 90. The rejection region was determined from Student's t distribution. The rejection region for our test was $t < -t_{\frac{\alpha}{2}}$ or $t > t_{\frac{\alpha}{2}}$ where $t_{\frac{\alpha}{2}}$ is based on 8 degrees of freedom. Hence, we can make one of the following conclusions:

- The first benchmark result is faster than the second benchmark result if $t < -2.306$
- The second benchmark result is faster than the first benchmark result if $t > 2.306$
- The sample evidence is insufficient to reject the null hypothesis at $\alpha = 0.05$

Our test results are statistically significant at the $\alpha = 0.05$ level of significance.

To allow the use of our statistical test, we had to assume that the population standard deviation for both samples was equal.

Table 90. Small-Sample Test Statistic for Simulation Benchmark *Small* Database Results

DBMS		Model Creation	Scenario Creation	Hour Run TS = 60	TS = 600	TS = 1800	TS = 3600	Version Creation	Map Creation	Report Creation
ObjectStore (*Remote*)	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	0.654	4.184	576.894	485.341	476.418	474.981	20.794	9.805	2.613
	Sample STD:	0.080	0.178	5.090	0.447	0.262	0.860	0.157	0.238	0.333
ObjectStore (*Local*)	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	0.633	4.285	580.294	490.979	482.764	477.213	21.168	10.243	2.999
	Sample STD:	0.100	0.621	0.710	2.508	3.810	1.462	0.716	0.329	0.854
	t =	0.367	-0.350	-1.479	-4.949	-3.716	-2.942	-1.141	-2.412	-0.942
ObjectStore (*Local*)	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	0.633	4.285	580.294	490.979	482.764	477.213	21.168	10.243	2.999
	Sample STD:	0.100	0.621	0.710	2.508	3.810	1.462	0.716	0.329	0.854
Non-Persistent	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	0.007	1.603	392.738	389.576	388.308	387.685	14.138	10.718	2.336
	Sample STD:	0.007	0.065	1.916	0.496	0.592	0.774	0.052	0.287	0.186
	t =	13.964	9.605	205.248	88.691	54.778	121.016	21.897	-2.433	1.696
ObjectStore (*Remote*)	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	0.654	4.184	576.894	485.341	476.418	474.981	20.794	9.805	2.613
	Sample STD:	0.080	0.178	5.090	0.447	0.262	0.860	0.157	0.238	0.333
Non-Persistent	N:	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
	Average:	0.007	1.603	392.738	389.576	388.308	387.685	14.138	10.718	2.336
	Sample STD:	0.007	0.065	1.916	0.496	0.592	0.774	0.052	0.287	0.186
	t =	18.015	30.456	75.714	320.708	304.332	168.710	89.990	-5.476	1.624

Appendix F. Benchmark Support Library

This appendix contains documentation and source code for the benchmark support library developed for this research. This library was developed to speed implementation of benchmark programs. The benchmark support library facilitated code reuse between benchmark program implementations. The benchmark support library consists of routines for measuring elapsed time, and routines for generating random numbers. The library consists of the following four modules:

- *The stopwatch module*
- *The dice module*
- *The expon module*
- *The pmmlcg U(0,1) pseudo-random number generator module*

Figure 71 shows a Coad/Yourdon OOA diagram of the benchmark support library (see Appendix A for a summary of the Coad/Yourdon OOA notation).

The code developed for the benchmark support library is in compliant with the coding standards developed for this research. These standards are described in Appendix G.

F.1 The Stopwatch

Elapsed time is the performance measurement used by the OO1 benchmark and the simulation benchmark. To allow accurate measurement of elapsed time, a C++ class called *stopwatch* was developed. The stopwatch class uses the system clock to obtain accurate timing. The benchmarks used in this research only require accuracy to about a millisecond, and to this level the system clock accuracy is reasonable. The basic timing components of this class were also used in the OO7 research project at the University of Wisconsin-Madison [7].

The stopwatch class is used by creating an instance of the class (usually through a declaration). Timing is started by calling the *start* method. Timing is stopped and recorded by calling the *stop* method.

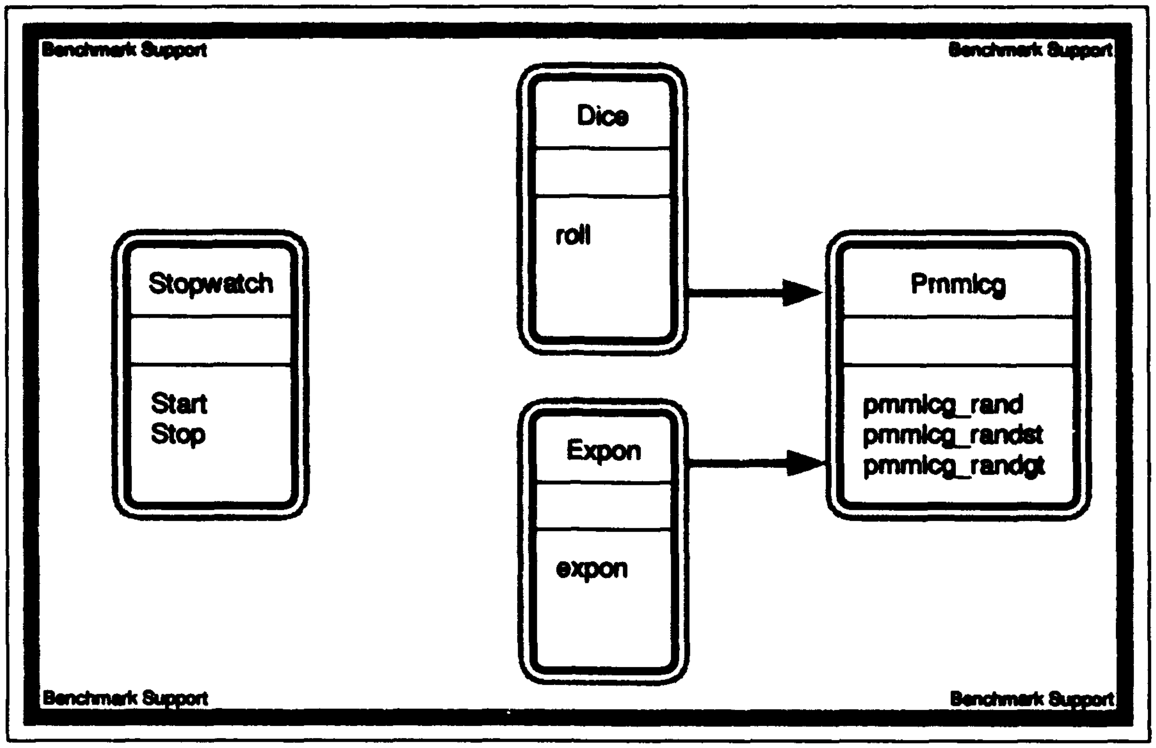

Figure 71. OOA Diagram for the Benchmark Support Library

F.1.1 stopwtch.hh. The source code for the file *stopwtch.hh* is listed below.

```
#ifndef __STOPWTCH_HH
#define __STOPWTCH_HH
/*

 #####
#     #   #####   ####   #####   #    #    ##     #####   ####   #    #
#           #    #    #  #    #  #    #   #  #      #    #    #  #    #
 #####      #    #    #  #    #  #    #  #    #     #    #       ######
      #     #    #    #  #####   # ## #  ######     #    #       #    #
#     #     #    #    #  #       ##  ##  #    #     #    #    #  #    #
 #####      #     ####   #       #    #  #    #     #     ####   #    #

 stopwtch.hh

 Air Force Institute of Technology
 Timothy J. Halloran
 20 May 1993

 NOTE: many of the techniques used in this support package came from
       the "Support.C" package created by Carey, DeWitt, and Naughton
       for the 007 benchmark (Objectivity implementation).
       [007 Benchmark COPYRIGHT (C) 1993 Carey DeWitt Naughton
        Madison, WI U.S.A. ALL RIGHTS RESERVED]

 Revisions:
```

```
 15 Jun 1993 -TJH- Modified to include <sys/time.h> to hide non-portable
                    system timing.
*/
#include<sys/time.h>

class stopwatch {
  struct timeval start_time;
  struct timeval stop_time;
  int clock_running;
public:
  stopwatch( );
  void start( );   // starts timing
  double stop( );  // stops timing, and returns the duration in seconds
};
#endif __STOPWTCH_HH
```

F.1.2 stopwtch.cc. The source code for the file *stopwtch.cc* is listed below.

```
/*

 #####
#     #   #####   ####   #####   #    #    ##     #####   ####   #    #
#           #    #    #  #    #  #    #   #  #      #    #    #  #    #
 #####      #    #    #  #    #  #    #  #    #     #    #       ######
      #     #    #    #  #####   # ## #  ######     #    #       #    #
#     #     #    #    #  #       ##  ##  #    #     #    #    #  #    #
 #####      #     ####   #       #    #  #    #     #     ####   #    #

 stopwtch.cc

 Air Force Institute of Technology
 Timothy J. Halloran
 20 May 1993

 NOTE: many of the techniques used in this support package came from
       the "Support.C" package created by Carey, DeWitt, and Naughton
       for the 007 benchmark (Objectivity implementation).
       [007 Benchmark COPYRIGHT (C) 1993 Carey DeWitt Naughton
        Madison, WI U.S.A. ALL RIGHTS RESERVED]

 Revisions:
 06 Jul 1993 -TJH- Changed all the debug output to use the "debug.hh" macros.
*/
#include<stdio.h>
#include<sys/time.h>
#include<debug.hh>
#include"stopwtch.hh"

#define TRUE   1
#define FALSE  0
//////////////////////////////////////////////////////////////////////
void stopwatch::start( )
{
  DEBUG_INIT( "STOPWATCH", "stopwatch::start" );
  DEBUG_OUT( "entering", 1 );
  if ( gettimeofday( &start_time, (struct timezone *)0 ) ) {
    fprintf( stderr, "ERROR[stopwatch::start] failed call to gettimeofday( )\n" );
  }
  clock_running = TRUE;
  DEBUG_OUT( sprintf( BUG, "seconds since Jan 1, 1970: %ld microseconds: %ld",
    start_time.tv_sec, start_time.tv_usec ), 2 );
}
//////////////////////////////////////////////////////////////////////
double stopwatch::stop( )
{
  DEBUG_INIT( "STOPWATCH", "stopwatch::stop" );
  DEBUG_OUT( "entering", 1 );
  // get the current time first (do error checking later)
  if ( gettimeofday( &stop_time, (struct timezone *)0 ) ) {
    fprintf( stderr, "ERROR[stopwatch::stop] failed call to gettimeofday( )\n" );
    return( 0.0 );
  }
  // make sure the clock was running
  if ( ! clock_running ) {
    // can't stop the stopwatch before you start it
    fprintf( stderr, "ERROR[stopwatch::stop] stop called before start\n" );
    return( 0.0 );
  }
  clock_running = FALSE;
  DEBUG_OUT( sprintf( BUG, "seconds since Jan 1, 1970: %ld microseconds: %ld",
    start_time.tv_sec, start_time.tv_usec ), 2 );
  // compute and return the duration
  double seconds = double( stop_time.tv_sec - start_time.tv_sec );
  double micro_seconds = double( stop_time.tv_usec - start_time.tv_usec );
  if ( micro_seconds < 0.0 ) {
    micro_seconds = 1000000.0 + micro_seconds;
    seconds--;
  }
  return( seconds + micro_seconds/1000000.0 );
}
//////////////////////////////////////////////////////////////////////
stopwatch::stopwatch( )
{
  DEBUG_INIT( "STOPWATCH", "stopwatch::stopwatch" );
  DEBUG_OUT( "entering", 1 );
  // when an object is first created, the clock is not running
  clock_running = FALSE;
}
//////////////////////////////////////////////////////////////////////
```

F.2 The Dice Module

The *dice* module is used in all the benchmark implementations in this thesis. It is used to create pseudo-random numbers for the benchmark programs.

F.2.1 dice.hh. The source code for the file *dice.hh* is listed below.

```
#ifndef __DICE_HH
#define __DICE_HH
```

```
/*

######
#     #     #     ####   ######
#     #     #    #     # #
#     #     #    #       #####
#     #     #    #       #
#     #     #    #     # #
######      #     ####   ######

 dice.hh

 Air Force Institute of Technology
 Timothy J. Halloran
 18 Aug 1993
*/
long roll( long low, long high, int stream );
#endif __DICE_HH
```

F.2.2 dice.cc. The source code for the file *dice.cc* is listed below.

```
/*

######
#     #     #     ####   ######
#     #     #    #     # #
#     #     #    #       #####
#     #     #    #       #
#     #     #    #     # #
######      #     ####   ######

 dice.cc

 Air Force Institute of Technology
 Timothy J. Halloran
 18 Aug 1993
*/
#include<stdio.h>
#include<pmmlcg.h>
#include<debug.hh>
#include"dice.hh"
//////////////////////////////////////////////////////////////////////
long roll( long low, long high, int stream )
{
  DEBUG_INIT( "DICE", "roll" );
  DEBUG_OUT( "entering", 1 );

  // return a random integer between "low" and "high" (inclusive)

  // check for a bad input parameter
  if ( high < low ) return low;
  // pmmlcg_rand( int stream ) returns non-negative floating-point
  //  values uniformly distributed over the interval (0,1)
  long result = low + (long)( pmmlcg_rand( stream ) * (double)( high + 1 - low ) );
  DEBUG_OUT( sprintf( BUG, "result %d from stream %d", result, stream), 2 );
  return( result );
}
////////////////////////////////////////////////////////////////////////
```

F.3 The Expon Module

The *expon* module is used by the simulation benchmark to produce random numbers with an exponential distribution.

F.3.1 expon.hh. The source code for the file *expon.hh* is listed below.

```
#ifndef __EXPON_HH
#define __EXPON_HH
/*

#######
#        #    #  #####    ####   #    #
#         #  #   #    #  #    #  ##   #
#####      ##    #    #  #    #  # #  #
#          ##    #####   #    #  #  # #
#         #  #   #       #    #  #   ##
#######  #    #  #        ####   #    #

 expon.hh

 Air Force Institute of Technology
 Timothy J. Halloran
 31 Aug 1993
*/
double expon( double mean, int stream );
#endif __EXPON_HH
```

F.3.2 expon.cc. The source code for the file *expon.cc* is listed below.

```
/*

#######
#        #    #  #####    ####   #    #
#         #  #   #    #  #    #  ##   #
#####      ##    #    #  #    #  # #  #
#          ##    #####   #    #  #  # #
#         #  #   #       #    #  #   ##
#######  #    #  #        ####   #    #

 expon.cc

 Air Force Institute of Technology
 Timothy J. Halloran
 31 Aug 1993
*/
#include<stdio.h>
#include<math.h>
#include<pmmlcg.h>
#include<debug.hh>
```

```
#include"expon.hh"
#include"dice.hh"
//////////////////////////////////////////////////////////////////////
double expon( double mean, int stream )
{
  DEBUG_INIT( "EXPON", "expon" );
  DEBUG_OUT( "entering", 1 );

  // return an exponential random variate with mean "mean"
  // pmmlcg_rand( int stream ) returns non-negative floating-point
  //  values uniformly distributed over the interval (0,1)
  return -mean * log( (double)pmmlcg_rand( stream ) );
}
//////////////////////////////////////////////////////////////////////
```

F.4 A U(0,1) Random Number Generator

Both the *dice* module and the *expon* module rely upon the generation of random numbers over U(0,1). The U(0,1) pseudo-random number generator used for all the programs in this research comes from the book *Simulation Modeling and Analysis* by Law and Kelton [24]. We decided to use this code after running some of the statistical tests for a U(1,0) random number generator described in by Law and Kelton. The UNIX system function *drand48( )*, and the random number generator described by Park and Miller in [30] (which was used by Cattell and Skeen in the original OO1 benchmark implementations [9]) were considered for use, but the generator described by Law and Kelton appeared to be superior.

F.4.1 pmmlcg.h. The source code for the file *pmmlcg.h* is listed below.

```
#ifndef __PMMLCG_H
#define __PMMLCG_H
/*

######  #     # #     # #        #####   #####
#     # ##   ## ##   ## #       #     # #     #
#     # # # # # # # # # #       #       #
######  #  #  # #  #  # #       #       #  ####
#       #     # #     # #       #       #     #
#       #     # #     # #       #     # #     #
#       #     # #     # #######  #####   #####

######
#     #    ##    #    #  #####    ####   #     #
#     #   #  #   ##   #  #    #  #    #  ##   ##
######   #    #  # #  #  #    #  #    #  # ## #
#   #    ######  #  # #  #    #  #    #  #    #
#    #   #    #  #   ##  #    #  #    #  #    #
#     #  #    #  #    #  #####    ####   #    #

#     #
##    #  #    #  #    #  #####   ######  #####
# #   #  #    #  ##  ##  #    #  #       #    #
#  #  #  #    #  # ## #  #####   #####   #    #
#   # #  #    #  #    #  #    #  #       #####
#    ##  #    #  #    #  #    #  #       #   #
#     #   ####   #    #  #####   ######  #    #

 #####
#     #  ######  #    #  ######  #####     ##    #####   ####   #####
#        #       ##   #  #       #    #   #  #     #    #    #  #    #
#  ####  #####   # #  #  #####   #    #  #    #    #    #    #  #    #
#     #  #       #  # #  #       #####   ######    #    #    #  #####
#     #  #       #   ##  #       #   #   #    #    #    #    #  #   #
 #####   ######  #    #  ######  #    #  #    #    #     ####   #    #

*/
#ifdef __cplusplus
extern "C" {
#endif
float pmmlcg_rand( int stream );
void pmmlcg_randst( long zset, int stream );
long pmmlcg_randgt( int stream );
#ifdef __cplusplus
}
#endif
#endif __PMMLCG_H
```

F.4.2 pmmlcg.c. The source code for the file *pmmlcg.c* is listed below.

```
/*

######  #     # #     # #        #####   #####
#     # ##   ## ##   ## #       #     # #     #
#     # # # # # # # # # #       #       #
######  #  #  # #  #  # #       #       #  ####
#       #     # #     # #       #       #     #
#       #     # #     # #       #     # #     #
#       #     # #     # #######  #####   #####

######
#     #    ##    #    #  #####    ####   #    #
#     #   #  #   ##   #  #    #  #    #  ##  ##
######   #    #  # #  #  #    #  #    #  # ## #
#   #    ######  #  # #  #    #  #    #  #    #
#    #   #    #  #   ##  #    #  #    #  #    #
#     #  #    #  #    #  #####    ####   #    #

#     #
##    #  #    #  #    #  #####   ######  #####
# #   #  #    #  ##  ##  #    #  #       #    #
#  #  #  #    #  # ## #  #####   #####   #    #
#   # #  #    #  #    #  #    #  #       #####
#    ##  #    #  #    #  #    #  #       #   #
#     #   ####   #    #  #####   ######  #    #

 #####
#     #  ######  #    #  ######  #####     ##     #####   ####   #####
#        #       ##   #  #       #    #   #  #      #    #    #  #    #
#  ####  #####   # #  #  #####   #    #  #    #     #    #    #  #    #
#     #  #       #  # #  #       #####   ######     #    #    #  #####
#     #  #       #   ##  #       #   #   #    #     #    #    #  #   #
 #####   ######  #    #  ######  #    #  #    #     #     ####   #    #

 Prime modulus multiplicative linear congruential generator
 Z[i] = (630360016 * Z[i-1]) (mod(pow(2,31) - 1)), based on Marse and
 Roberts' portable FORTRAN random-number generator UNIRAN.  Multiple
 streams (100) are supported, with seeds spaced 100,000 apart.
 Throughout, input argument "stream" must be an int giving the desired
 stream number.  The header file rand.h must be included in the calling
 program (#include "pmmlcg.h") before using these functions.

 Usage: (three functions)

 1. To obtain the next U(0,1) random number from the stream "stream,"
    execute
        u = pmmlcg_rand( stream );
    where pmmlcg_rand is a float function.  The float variable u will
    contain the next random number.

 2. To set the seed for the stream "stream" to a desired value zset,
    execute
        pmmlcg_randst( zset, stream );
    where pmmlcg_randst is a void function and zset must be a long set
    to the desired seed, a number between 1 and 2147483646 (inclusive).
    Default seeds for all 100 streams are given in the code.

 3. To get the current (most recently used) integer in the sequence
    being generated for stream "stream" into the long variable zget,
    execute
        zget = pmmlcg_randgt( stream );
    where pmmlcg_randgt is a long function.
*/
#include"pmmlcg.h"

/* define the constants */
#define MODLUS 2147483647
#define MULT1       24112
#define MULT2       26143

/* set the default seeds for all 100 streams */
static long zrng[] =
{         0,
 1973272912,  281629770,   20006270, 1280689831, 2096730329,
 1933576050,  913566091,  246780520, 1363774876,  604901985,
 1511192140, 1259851944,  824064364,  150493284,  242708531,
   75253171, 1964472944, 1202299975,  233217322, 1911216000,
  726370533,  403498145,  993232223, 1103205531,  762430696,
 1922803170, 1385516923,   76271663,  413682397,  726466604,
  336157058, 1432650381, 1120463904,  595778810,  877722890,
 1046574445,   68911991, 2088367019,  748545416,  622401386,
 2122378830,  640690903, 1774806513, 2132545692, 2079249579,
   78130110,  852776735, 1187867272, 1351423507, 1645973084,
 1997049139,  922510944, 2045512870,  898585771,  243649545,
 1004818771,  773686062,  403188473,  372279877, 1901633463,
  498067494, 2087759558,  493157915,  597104727, 1530940798,
 1814496276,  536444882, 1663153658,  855503735,   67784357,
 1432404475,  619691088,  119025595,  880802310,  176192644,
 1116780070,  277854671, 1366580350, 1142483975, 2026948561,
 1053920743,  786262391, 1792203830, 1494667770, 1923011392,
 1433700034, 1244184613, 1147297105,  539712780, 1545929719,
  190641742, 1645390429,  264907697,  620389253, 1502074852,
  927711160,  364849192, 2049576050,  638580085,  547070247
};
/**********************************************************************/
float pmmlcg_rand( int stream )
{
  long zi, lowprd, hi31;

  /* generate the next random number */
  zi     = zrng[stream];
  lowprd = ( zi & 65535 ) * MULT1;
  hi31   = ( zi >> 16 ) * MULT1 + ( lowprd >> 16 );
  zi     = ( ( lowprd & 65535 ) - MODLUS ) +
    ( ( hi31 & 32767 ) << 16 ) + ( hi31 >> 15 );
  if ( zi < 0 ) zi += MODLUS;
  lowprd = ( zi & 65535 ) * MULT2;
  hi31   = ( zi >> 16 ) * MULT2 + ( lowprd >> 16 );
  zi     = ( ( lowprd & 65535 ) - MODLUS ) +
    ( ( hi31 & 32767 ) << 16 ) + ( hi31 >> 15 );
  if ( zi < 0 ) zi += MODLUS;
  zrng[stream] = zi;
  return ( ( zi >> 7 | 1 ) + 1 ) / 16777216.0;
}
/**********************************************************************/
void pmmlcg_randst( long zset, int stream )
{
  /* set the current zrng for the stream "stream" to zset */
  zrng[stream] = zset;
}
/**********************************************************************/
long pmmlcg_randgt( int stream )
{
  /* return the current zrng for the stream "stream" */
  return zrng[stream];
}
/**********************************************************************/
```

Appendix G. Code Style Guide

An important issue when writing code in three different object-oriented DBMSs is to ensure that the code is understandable. This appendix describes the style rules which were followed during the development of source code for this research. The purpose for these style rules was to provide consistency across the large amount of source code which was written during this research.

G.1 Code Indentation and Spacing

All the source code used two space indentation between logical levels. No tabs were used in any of the source code developed for this research. Whitespace in the source code, both vertical and horizontal, was used to make the code easier to read. Figure 72 provides a typical example of good indentation and spacing.

G.2 Naming Conventions

G.2.1 Variable and Function Names. Variable and function names contained only small letters. Words in the names were separated by an underscore. The following would have been valid variable or function names: `start`, `micro_seconds`. The following would *not* have been valid variable or function names: `Start`, `MicroSeconds`. Invalid names were only used if the interface to an external library required them. For example, ObjectStore uses the name `os_Set` for a set when it is parameterized, and `os_set` when it is not. Abbreviations were used in some names. All the abbreviations used in our development are defined in Table 91.

G.2.2 Constant Names. Constant names contained only capital letters. Words in constant names were separated by an underscore. Capital letters were also used for C++ enumerated types. The following would have been valid constant names: `TRUE`, `TOP_OF_STACK`. The following would *not* bave been valid constant names: `True`, `top_of_stack`. Invalid names were only used if the interface to an external library required them. Abbreviations were used in some constant names. All the abbreviations used in our development are defined in Table 91.

```
//////////////////////////////////////////////////////////////////////
void stopwatch::start( )
{
  DEBUG_INIT( "STOPWATCH", "stopwatch::start" );
  DEBUG_OUT( "entering", 1 );

  if ( gettimeofday( &start_time, (struct timezone *)0 ) ) {
    fprintf( stderr, "ERROR[stopwatch::start] failed call to gettimeofday( )\n" );
  }
  clock_running = TRUE;

  DEBUG_OUT( sprintf( BUG, "seconds since Jan 1, 1970: %ld microseconds: %ld",
    start_time.tv_sec, start_time.tv_usec ), 2 );
}
//////////////////////////////////////////////////////////////////////
```

Figure 72. Example of Source Code Indentation and Spacing

G.3 Comments

Source code comments were not used to represent things which are obvious from the source code.

G.3.1 Module/Header Comments. The top of each module (a "`.c`" or "`.cc`" file) and header file (a "`.h`" or "`.hh`" file) contains a comment block. The purpose of this block was to identify the source file and inform a reader of any important information regarding the entire module. All revisions made to a source file are listed in the comment block. Figure 73 shows an example of a comment block for a source file which had one revision.

G.3.2 Inline Comments. The C++ comment indicator (// *a comment* `<eol>`) was preferred to the C comment indicators (/* *a comment* */) for inline comments. Obviously, if the file was designed to be compiled by a C compiler this convention was not followed.

G.3.3 Function Headers and Separator Comments. Each function was not given a comment block but a separator was be used to make it obvious to a reader that a new function had been started. The function separator consisted of a line of 70 "/" characters.

```
/*

 #####
#     #   #####   ####   #####   #    #    ##     #####   ####   #    #
#           #    #    #  #    #  #    #   #  #      #    #    #  #    #
 #####      #    #    #  #    #  #    #  #    #     #    #       ######
      #     #    #    #  #####   # ## #  ######     #    #       #    #
#     #     #    #    #  #       ##  ##  #    #     #    #    #  #    #
 #####      #     ####   #       #    #  #    #     #     ####   #    #

 stopwtch.cc

 Air Force Institute of Technology
 Timothy J. Halloran
 20 May 1993

 NOTE: many of the techniques used in this support package came from
       the "Support.C" package created by Carey, DeWitt, and Naughton
       for the 007 benchmark (Objectivity implementation).
       [007 Benchmark COPYRIGHT (C) 1993 Carey DeWitt Naughton
        Madison, WI U.S.A. ALL RIGHTS RESERVED]

 Revisions:
 06 Jul 19[illegible] -TJH- Changed all the debug output to use the "debug.hh" macros.
*/
```

Figure 73. Source File Comment Block Example

Table 91. Standard Abbreviations

Abbreviation	Full Name
app	application
db	database
cb	callback
config	configuration
fmt	format
id	identifier
it	Itasca
num	number
ma	Matisse
max	maximum
min	minimum
misc	miscellaneous
msg	message
os	minimum
pos	position
prev	previous
proc	procedure
sim	ObjectStore
str	string
temp	temporary

For a C program a line of "*" characters inside a comment was used (the total separator line still consisted of 70 characters).

G.4 Error Output

All error output was standardized. Error output was broken into two types: *errors* and *warnings*. The difference between the two is that a program will exit if an error occurs, but continues when a warning occurs. For all error output, the name of the function (or class method) where the problem occurred appears inside brackets after the word "ERROR" or "WARNING". Figure 74 provides some examples of error output.

G.5 Debug Output

All program debug output was standardized by using the macros in the file *debug.hh* (listed in the next section). This macro package was converted from a macro package used

```
WARNING[main] the random seed is 0
WARNING[create_connections] invalid query result
ERROR[stopwatch::start] failed call to gettimeofday( )
```

Figure 74. Examples of Program Error Output

```
[ool::ool(ool.cc:326)] entering
[dice::dice(dice.cc:31)] entering
[ool::forward_traversal(ool.cc:128)] entering
[stopwatch::stopwatch(stopwtch.cc:80)] entering
[stopwatch::start(stopwtch.cc:37)] entering
[stopwatch::start(stopwtch.cc:44)] seconds since Jan 1, 1970: 742061664
[dice::roll(dice.cc:38)] entering
[dice::roll(dice.cc:44)] result 324
[part::forward_traversal(part.cc:57)] entering
[part::forward_traversal(part.cc:59)] part id    324 (level 0)
[part::forward_traversal(part.cc:57)] entering
[part::forward_traversal(part.cc:59)] part id    325 (level 1)
[part::forward_traversal(part.cc:57)] entering
[part::forward_traversal(part.cc:59)] part id    406 (level 2)
```

Figure 75. Example of Debug Output

by Microsoft [27]. The macros were intended to provide run-time tracing for programs developed for the *Microsoft Windows* environment and were modified so they could be used for this research. The advantages of the macro package are the following: it does not require a debugger, it is controlled by environment variables, and its debug code can be compiled out when debugging is finished.

Debug output consists of two parts: a *location* and a *message*. The location consists of the function name where the debug was output, the name of the source file which contains the function, and the line number inside the source file where the debug output line was located. This information is inclosed in brackets. The message contains the debug information which is to be output and is allowed to be in any format desired. Figure 75 shows an example of program debug output.

G.5.1 debug.hh. The source code for the file *debug.hh* is listed below.

```
#ifndef __DEBUG_HH
#define __DEBUG_HH
```

```
/*
######
#     # ###### #####  #    #  ####
#     # #      #    # #    # #    #
#     # #####  #####  #    # #
#     # #      #    # #    # #  ###
#     # #      #    # #    # #    #
######  ###### #####   ####   ####

 debug.h

 Air Force Institute of Technology
 Timothy J. Halloran
 06 Jul 1993

 NOTE: the basic ideas in this file came from "Debugging Without Debuggers"
       Microsoft Systems Journal Vol. 8, No. 4, April 1993, Pages 52-55.

 Some useful debugging macros which are controlled by environment variables.

 Revisions:
 26 Jul 1993 -TJH- Changed the DEBUG_INIT macro to locally declare the
                   "location" variable.  This variable was being overwritten
                   by the debug code inside functions which were called from
                   inside a function using debug output.
*/
#ifdef DEBUG
#include<stdio.h>
#include<stdlib.h>
static int debug_level = 0;
static char BUG[300];  /* a buffer for use by "sprintf( )" */

/*
  DEBUG_INIT (macro)

  env              (char*) Contains the name of the environment variable
                           which turns tracing on or off for the function.
  current_location (char*) Application defined location information (such
                           as the name of the current function being
                           traced).
*/
#define DEBUG_INIT( env, current_location )  \
  char *location = current_location;         \
{                                            \
  if ( getenv( env ) != NULL )               \
    debug_level = atoi( getenv( env ) );     \
  else                                       \
    debug_level = 0;                         \
}

/*
  DEBUG_OUT (macro)

  out              (char*) The message to be output.
  level            (int)   The trace level at which this message is output.
```

```
*/
#define DEBUG_OUT( out, level ) {                          \
  if ( debug_level >= level )                              \
    fprintf( stderr, "[%s(%s:%d)] %s\n", location,  \
      __FILE__, __LINE__, out );                           \
}

#else

/* compile out all trace instructions, if DEBUG is not defined */
#define DEBUG_INIT( env, current_location )
#define DEBUG_OUT( out, level )

#endif DEBUG
#endif __DEBUG_HH
```

Appendix H. Benchmark Source Code

The source code for all the benchmark implementations created for this research is available in AFIT Technical Report AFIT/EN-TR-93-09 [16]. The source code was not included in this thesis due to its size. The technical report contains source code for the following benchmark implementations:

- Itasca DBMS implementation of the OO1 benchmark
- Matisse DBMS implementation of the OO1 benchmark
- ObjectStore DBMS implementation of the OO1 benchmark
- ObjectStore DBMS implementation of the Simulation benchmark
- A non-persistent implementation of the Simulation benchmark (in the C++ programming language)

Bibliography

1. Anon, et al. "A Measure of Transaction Processing Power," *Datamation*, *31*(7):112–118 (April 1985).
2. Atkinson, Malcom, et al. *The Object-Oriented Database System Manifesto*. Technical Report 30-89, LeChesnay, France: GIP ALTAIR, September 1989.
3. Banerjee, Jay, et al. "Queries in an Object-Oriented Databases," *Proceedings 4th IEEE DEC*, 31–38 (1988).
4. Barry, Douglas K. "ODBMS Feature Listing," *Object Magazine*, 48–53 (February 1993).
5. Borland International. *Quattro Pro for Windows User's Guide*. Scotts Valley, CA: Borland International, Inc., 1993.
6. Brown, Randy E. and William K. McQuay. "The Joint Modeling and Simulation System," *Journal of Electronic Defense* (September 1992).
7. Carey, Michael J., et al. *The OO7 Benchmark*. Technical Report, University of Wisconsin-Madison, April 12, 1993. available via anonymous ftp from cs.wisc.edu.
8. Cattell, R.G.G. *Object Data Management Object-Oriented and Extended Relational Database Systems*. Reading, MA: Addison-Wesley Publishing Company, 1991.
9. Cattell, R.G.G. and J. Skeen. "Object Operations Benchmark," *ACM Transactions on Database Systems*, *17*(1):1–31 (March 1992).
10. Coad, Peter and Edward Yourdon. *Object-Oriented Analysis* (Second Edition). Eaglewood Cliffs, NJ: Prentice Hall, Inc., 1991.
11. Coad, Peter and Edward Yourdon. *Object-Oriented Design*. Eaglewood Cliffs, NJ: Prentice Hall, Inc., 1991.
12. Date, C. J. *An Introduction to Database Systems* (Fifth Edition), *I*. Reading, MA: Addison-Wesley Publishing Company, 1991.
13. Garza, Jorge F. and Won Kim. "Transaction Management in an Object-Oriented Database System," *Proceedings ACM SIGMOD*, 37–45 (1988).
14. Gray, Jim, editor. *The Benchmark Handbook for Database and Transaction Processing Systems*. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1991.
15. Gupta, Rajiv and Ellis Horowitz, editors. *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD*. Englewood Cliffs, NJ 07632: Prentice Hall, 1991.
16. Halloran, Timothy J. *Source Code for the OO1 Benchmark and the AFIT Simulation Benchmark*. Technical Report AFIT/EN-TR-93-09, Air Force Institute of Technology (AETC), December 1993.
17. Heller, Dan. *Motif Programming Manual for OSF/Motif Version 1.1*. Sebastopal, CA: O'Reilly & Associates, Inc., 1991.
18. Hennessy, John L. and David A. Patterson. *Computer Architecture A Quantitative Approach*. San Mateo, CA 94403: Morgan Kaufmann Publishers, Inc., 1990.

19. Intellitic International. *Object-Oriented Services Programmer's Reference for Matisse 2.2*. Paris: Intellitic International SNC, February 1993.

20. Itasca Systems. *C++ API User Manual for Release 2.2*. Minneapolis, MN: Itasca Systems, Inc., 1993.

21. J-MASS Design Team. *SSE Architectural Design Team Report*. Wright-Patterson AFB, OH: J-MASS Program Office, 1992.

22. Kim, Won and Hong-Tai Chou. "Versions of Schema for Object-Oriented Databases," *Proceedings of the 14th VLDB Conference*, 148–159 (1988).

23. Kim, Won, et al. "Architecture of the ORION Next-Generation Database System," *IEEE Transactions on Knowledge and Data Engineering*, *2*(1):109–124 (March 1990).

24. Law, Averill M. and W. David Kelton. *Simulation Modeling and Analysis* (Second Edition). New York: McGraw-Hill, Inc., 1991.

25. Mathias, Karl S. *Integration and Enhancement of the Saber Wargame*. MS thesis, AFIT/GCS/ENG/93D-15, Air Force Institute of Technology (AETC), December 1993.

26. McClave, James T. and P. George Benson. *Statistics for Business and Economics*. San Francisco, CA: Dellen Publishing Company, 1991.

27. Mirho, Charles. "Debugging Without Debuggers," *Microsoft Systems Journal*, *8*(2):52–55 (April 1993).

28. Object Design. *User Guide ObjectStore Release 2.0 for UNIX Systems*. Burlington, MA: Object Design, Inc., October 1992.

29. Open Software Foundation. *OSF/Motif Style Guide*. MA: Open Software Foundation, Inc., January 1992.

30. Park, Stephen K. and Keith W. Miller. "Random Number Generators: Good Ones are Hard to Find," *Communications of the ACM*, *31*(10):1192–1201 (October 1988).

31. Pidd, Michael, editor. *Computer Modelling for Discrete Simulation* (Second Edition). Chichester: John Wiley & Sons Ltd., 1989.

32. Pidd, Michael. "Guidelines for the design of data driven generic simulators for specific domains," *Simulation*, *59*(4):237–243 (October 1992).

33. Richman, Dan. "Standard Raises Hopes," *Open Systems Today*, 51 (25 October 1993).

34. Rooks, Michael. "A User-Centered Paradigm for Interactive Simulation," *Simulation*, *60*(3):168–177 (March 1993).

35. Rubenstein, W.B., et al. "Benchmarking Simple Database Operations," *Proceedings ACM SIGMOD*, 387–394 (May 1987).

36. Rumbaugh, James, et al. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

37. Stonebraker, Michael, et al. *Third-Generation Database System Manifesto*. Memorandum UCB/ERL M90/28, College of Engineering, University of California, Berkeley, CA 94720: Electronics Research Laboratory, April 1990.

38. Transaction Processing Performance Council, "TPC Benchmark C Standard Specification," 13 August 1992.

39. Transaction Processing Performance Council, "TPC Press Backgrounder," 1992.

40. Woyna, Mark A., et al. "Modeling Battlefield Sensor Environments with an Object Database Management System," *ACM SIGMOD Record*, *22*(3):499–501 (June 1993).

Vita

Captain Timothy J. Halloran was born on June 14, 1965 in Westfield, Massachusetts. He graduated from Westfield High School in 1983. He then attended the United States Air Force Academy and graduated with a degree in Computer Science in 1987. His first permanent duty assignment was to the Air Force Wargaming Center at Maxwell AFB, Alabama. As a Senior Simulation Analyst, he specialized in the design and development of computer wargame simulations which supported the schools of Air University. He entered the School of Engineering, Air Force Institute of Technology in May, 1992.

Permanent address: 12 Longfellow Drive
Wilbraham, Massachusetts 01095

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
December, 1993

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE
PERFORMANCE MEASUREMENT OF THREE COMMERCIAL OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS

5. FUNDING NUMBERS

6. AUTHOR(S)
Timothy J. Halloran, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/93D-12

9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Capt Rick Painter
2241 Avionics Circle, Suite 16
WL/AAWA-1 BLD 620
Wright-Patterson AFB, OH 45433-7765
(513) 255-4429 or DSN 785-4429

10. SPONSORING MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The goal of this thesis was to study the performance of three commercial object-oriented database management systems. The commercial systems studied included: Itasca, sold by Itasca Systems Incorporated; Matisse, sold by Intellitic International; and ObjectStore, sold by Object Design Incorporated. To examine performance of these database management systems two benchmarks were run: the OO1 benchmark and a new AFIT Simulation benchmark. The OO1 benchmark was designed, implemented, and run on all three database management systems. ObjectStore was our top performer on all configurations of the OO1 benchmark. The AFIT Simulation benchmark was designed, implemented, and run on the ObjectStore database management system. A non-persistent version of the benchmark was also created in the C++ programming language. There was minimal performance overhead incurred due to the use of ObjectStore, especially when compared to the functional benefits gained. We concluded that there are major differences between the performance levels offered in current commercial object-oriented database management systems. We also concluded that a programming language interface to an object-oriented database management system should not be middle ground. Either it should be closely tied to a specific language or not tied to a specific language at all.

14. SUBJECT TERMS
Database Benchmarks, Object-Oriented Databases, Simulation

15. NUMBER OF PAGES
260

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL